

DTIC FILE COPY

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A219 806



THESIS

CIRCUIT RECOGNITION OF VLSI LAYOUTS

by

Joel V. Swisher

September 1989

Thesis Advisor:

Chyan Yang

Approved for public release; distribution is unlimited

DTIC
S ELECTE D
MAR 29 1990
B

90 00 70 101

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

| REPORT DOCUMENTATION PAGE | | | | Form Approved OMB No. 0704-0188 | |
|---|--|--|--|--|--|
| 1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED | | | 1b RESTRICTIVE MARKING | | |
| 2a SECURITY CLASSIFICATION AUTHORITY | | | 3 DISTRIBUTION AVAILABILITY OF REPORT Approved for public release; distribution is unlimited | | |
| 2b DECLASSIFICATION/DOWNGRADING SCHEDULE | | | | | |
| 4 PERFORMING ORGANIZATION REPORT NUMBER(S) | | | 5 MONITORING ORGANIZATION REPORT NUMBER(S) | | |
| 6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School | | 6b OFFICE SYMBOL (If applicable) 62 | | 7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School | |
| 6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000 | | | 7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000 | | |
| 8a NAME OF FUNDING SPONSORING ORGANIZATION | | 8b OFFICE SYMBOL (If applicable) | | 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER | |
| 8c ADDRESS (City, State, and ZIP Code) | | | 10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. PROJECT NO. TASK NO. WORK UNIT ACROSSING | | |
| 11 TITLE (Include Security Classification) CIRCUIT RECOGNITION OF VLSI LAYOUTS | | | | | |
| 12 PERSONAL AUTHOR(S) SWISHER, Joel V. | | | | | |
| 13a TYPE OF REPORT Master's Thesis | | 13b TIME COVERED FROM TO | | 14 DATE OF REPORT (Year, Month, Day) 1989 September | |
| 15 PAGE COUNT 114 | | | | | |
| 16 SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government | | | | | |
| 17 GSAF CODES FIELD OFFICE SUB GROUP | | | 18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) VLSI design; Circuit Verification; VLSI circuit design and timing verifier; simulation files; Circuit Recognition | | |
| 19 ABSTRACT (Continue on reverse if necessary and identify by block number) The design process of a very large scale integrated (VLSI) circuit is time consuming, with design verification and timing analysis being two of the most tedious stages. The development of a computer-aided design (CAD) tool that verifies circuit design and timing will reduce the design time. The primary contribution of this thesis is to provide an initial tool that will assist VLSI designers with the verification of a circuit's design. This tool is the first of several modular programs which will give the designer the capability to quickly and accurately verify a VLSI circuit's design and timing. The primary goal of this thesis is to develop an algorithm that will recognize different elements within the simulation file of a Complementary Metal Oxide Silicon (CMOS) circuit. Several simulation files were obtained using Magic which is a layout editing system developed at the | | | | | |
| 20 DISTRIBUTION AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED <input type="checkbox"/> CONFIDENTIAL <input type="checkbox"/> SECRET | | | 21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED | | |
| 22a NAME OF RESPONSIBLE INDIVIDUAL YANG, Chyan | | | 22b TELEPHONE (Include Area Code) 408-646-2266 | | |
| | | | 22c MAILING ADDRESS 61Ya | | |

DD Form 1473, JUN 86

S/N 0102-11-011-6000

UNCLASSIFIED

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. continued

University of California, Berkeley. These simulation files were analyzed and a C program was written that would accomplish circuit recognition. Results demonstrate that recognition of not only transistors, inverters, and passgates is possible, but also complex elements. A section is provided that describes possible uses for this algorithm.

| | |
|--------------------|--|
| Accession For | |
| NTIS GRA&I | <input checked="checked" type="checkbox"/> |
| DTIC TAB | <input type="checkbox"/> |
| Unannounced | <input type="checkbox"/> |
| Justification | |
| By | |
| Distribution/ | |
| Availability Codes | |
| Dist | Avail and/or Special |
| A-1 | |



S H 0102-LE 014-8601

UNCLASSIFIED

Approved for public release; distribution is unlimited

Circuit Recognition of VLSI Layouts

by

Joel V. Swisher
Captain, USA.
B.S., USMA, 1980

Submitted in partial fulfillment of the
requirements for the degree of

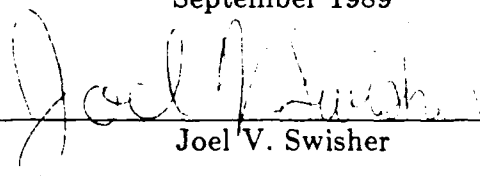
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

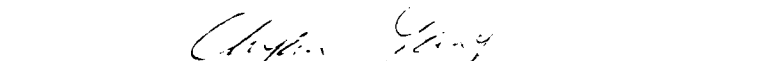
NAVAL POSTGRADUATE SCHOOL


September 1989

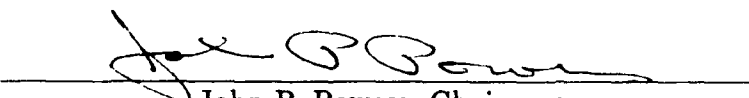
Author:


Joel V. Swisher

Approved by:


Chyan Yang, Thesis Advisor


Jon T. Butler, Second Reader


John P. Powers, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

The design process of a very large scale integrated (VLSI) circuit is time consuming, with design verification and timing analysis being two of the most tedious stages. The development of a computer-aided design (CAD) tool that verifies circuit design and timing will reduce the design time. The primary contribution of this thesis is to provide an initial tool that will assist VLSI designers with the verification of a circuit's design. This tool is the first of several modular programs which will give the designer the capability to quickly and accurately verify a VLSI circuit's design and timing.

The primary goal of this thesis is to develop an algorithm that will recognize different elements within the simulation file of a Complementary Metal Oxide Silicon (CMOS) circuit. Several simulation files were obtained using Magic which is a layout editing system developed at the University of California, Berkeley. These simulation files were analyzed and a C program was written that would accomplish circuit recognition. Results demonstrate that recognition of not only transistors, inverters, and passgates is possible, but also complex elements. A section is provided that describes possible uses for this algorithm.

TABLE OF CONTENTS

| | | |
|------|--|----|
| I. | INTRODUCTION | 1 |
| | A. BACKGROUND | 1 |
| | B. SCOPE OF THE THESIS INVESTIGATION | 3 |
| | C. THESIS OUTLINE | 3 |
| II. | INTRODUCTION TO THE SIMULATION FILE. | 5 |
| | A. MAGIC | 5 |
| | B. SIMULATION FILES | 6 |
| III. | EXAMINING .SIM FILES OF KNOWN CMOS CIRCUITS. | 8 |
| | A. MANUAL RECOGNITION | 8 |
| | B. THE ALGORITHM | 12 |
| | C. IMPLEMENTATION OF THE ALGORITHM | 15 |
| | D. ALGORITHM ACCOMPLISHMENTS | 18 |
| IV. | DEVELOPING CONNECTIVITY TO MORE TRANSISTORS. | 19 |
| | A. LEVEL1 MANUAL RECOGNITION | 19 |
| | B. LEVEL2 MANUAL RECOGNITION | 23 |
| | C. LEVEL1 ALGORITHM | 23 |
| | D. LEVEL2 ALGORITHM | 26 |
| | E. IMPLEMENTATION OF HIGHER LEVEL ALGORITHMS | 26 |
| | F. HIGHER LEVEL ALGORITHM ACCOMPLISHMENTS | 29 |
| V. | CONCLUSIONS AND RECOMMENDATIONS. | 30 |
| | A. THE RECOGNITION ALGORITHM | 30 |
| | B. FUTURE RESEARCH | 30 |

| | |
|---|-----|
| APPENDIX A: PROGRAM LISTINGS | 32 |
| A. GLOBAL VARIABLES | 32 |
| B. MAIN PROGRAM | 38 |
| C. TRANSISTOR LIST FUNCTION | 41 |
| D. INVERTER AND PASSGATE RECOGNITION FUNCTION . . | 45 |
| E. LEVEL 1 RECOGNITION FUNCTION | 55 |
| F. LEVEL 2 RECOGNITION FUNCTION | 69 |
| G. PRINT FUNCTIONS | 91 |
| H. FUNCTION TO READ THE INPUT FILE | 95 |
| I. FUNCTIONS TO CREATE DATA STRUCTURES | 96 |
| J. ADDITIONAL FUNCTIONS | 99 |
| REFERENCES | 100 |
| INITIAL DISTRIBUTION LIST | 102 |

LIST OF TABLES

| | | |
|-----|---|----|
| 3.1 | Comparisons of Program Speeds on the VAX 11/785 | 17 |
| 3.2 | Comparisons of Program Speeds on ISI | 17 |

LIST OF FIGURES

| | | |
|-----|--|----|
| 3.1 | A CMOS inverter. | 9 |
| 3.2 | A .sim file for an inverter. | 9 |
| 3.3 | Transistor lines in a .sim file. | 10 |
| 3.4 | A .sim file for a passgate. | 11 |
| 3.5 | A CMOS passgate. | 12 |
| 3.6 | A header line for a .sim file. | 13 |
| 4.1 | Two Level1 Category 1 Devices | 20 |
| 4.2 | A Level1 Category 2 Device | 20 |
| 4.3 | Connectivity with Vdd | 21 |
| 4.4 | No Connectivity with GND | 22 |

ACKNOWLEDGMENT

I would like to express my gratitude and appreciation to the faculty and staff of the Electrical and Computer Engineering department for providing me with the opportunity and encouragement to explore many exciting facets of electrical engineering. I would like to offer special thanks to Professor Chyan Yang for providing the necessary guidance and direction in the formulation of this document. I also wish to thank Professor Jon T. Butler for his valued assistance as my second reader.

Finally, I am most grateful to my wife Cindi for her patience, understanding, and support during my studies; And to my children: Matthew, Daniel, and Erika for just being mine .

I. INTRODUCTION

A. BACKGROUND

This thesis is the result of a larger effort to develop a fast very large scale integrated (VLSI) circuit design and timing verifier. The design of VLSI circuits is a time consuming process that includes four stages: logic design, layout design, circuit simulation, and circuit timing [1][2]. Development of a quick layout design and timing verifier could reduce the design time.

During the design process of a VLSI circuit, designers translate their design from a circuit specification into the circuit's schematic diagram. Then, a geometric layout is developed that is used to generate the masks that are used in the circuit's fabrication. This translation process is the bridge between the stages of logic design and layout design. It can become extremely time consuming because of mistakes which would lead to several iterations within the translation process. Identifying all mistakes prior to the mask generation is critical to the designers. Thus, making a timely design verification before submitting the mask layout for processing is an important and necessary step within this translation process.

While design verification is important, so also is the circuit's performance verification. A designer may produce a functionally correct geometric layout for mask generation. Unfortunately, if the layout does not meet circuit timing specifications, then the layout is not useful. An accurate timing verifier will validate all the paths within a circuit to ensure that the delays are within the proper limits of tolerance.

Most design verification of VLSI circuits is accomplished through switch-level simulation, because device level simulation (such as SPICE [3]) is too time consuming. If the circuit is incorrectly designed, the results from the switch-level simulator may not provide enough information to identify the fault. When this occurs, design verification is accomplished through visual means after enlarging the layout so that visual verification is possible. Again this process is time consuming.

Most timing verifiers treat a circuit as a graph of the circuit nodes. These verifiers enumerate all paths from the input to the output and compute the timing delays for each. This technique tends to consume a great deal of computer time.

Without a timely and accurate circuit design and timing verifier, costs can soar because of inefficient verification techniques as well as the fabrication of improper circuits. There have been several studies made in an attempt to reduce the time required to perform certain stages of the VLSI design process. Most of these studies have concentrated on only one stage of the design process. Some studies have concentrated only on layout design [4][5]. Still others have concentrated solely on circuit timing [6][7][8].

A VLSI circuit design and timing verifier uses an algorithm that reduces the time of the entire design process of integrated circuits. Regularity within a VLSI design helps in the identification of individual components within a given circuit. By categorizing these components into like elements and then comparing these elements with the proper amount and type of elements required for the actual design, a quick verification can be performed. For example, if the designer knows that his design contains 100,012 transistors, and 9,821 inverters, the algorithm can confirm these amounts, and then a quick verification has been performed. Furthermore, by using components of like elements in timing analysis rather than using single circuit nodes, the time required for performance verification can also be significantly reduced. This

reduction is achieved because the total number of elements (components versus single nodes) along a given path is smaller. Thus, the time required to compute timing delays along that path will decrease.

The research performed within this thesis concentrates on the portion of the algorithm that will be used to verify the layout design. This is accomplished by verifying what is actually placed on the VLSI layout. The verification is achieved by analyzing simulation files. These files are discussed in Chapter II.

B. SCOPE OF THE THESIS INVESTIGATION

The primary goal of this thesis is to develop an algorithm that will recognize different elements within a Complementary Metal Oxide Silicon (CMOS) circuit. The algorithm uses a simulation file as input. It performs a search of the simulation file identifying all passgates and inverters. The algorithm also connects the remaining transistors within the circuit as well as reforming these transistors to produce the correct CMOS circuit.

This goal was met by first developing an algorithm to recognize a passgate and an inverter from a simulation file. The algorithm was then expanded to incorporate the connection of the remaining transistors within the circuit. Finally, the algorithm was modified to connect the individual circuits.

C. THESIS OUTLINE

Chapter II provides an introduction to Magic and its application features. A discussion of its hierarchical extractor and the method of producing simulation files is addressed.

The examination of simulation files of known CMOS circuits is presented in Chapter III. The development of a manual recognition algorithm is discussed, as

well as the implementation and testing of a successful automatic recognition algorithm for transistors, passgates, and inverters.

Chapter IV discusses the levels of the recognition process. The development of the two algorithms which connect the remaining transistors within the circuit and reform these connected components to produce the correct CMOS circuit is presented.

Chapter V summarizes the key results of this thesis. This chapter includes a discussion on possible uses within the circuit design arena.

II. INTRODUCTION TO THE SIMULATION FILE.

The simulation file is used as input data for a computer program that is being designed to recognize features within a Complementary Metal Oxide Silicon (CMOS) layout. Since the simulation file is the basic building block for this research, it is important to discuss how this file is obtained. Magic will be discussed first, because the simulation file is created using Magic's tools.

A. MAGIC

Magic is a computer-aided design tool developed to aid the CMOS designer with integrated circuit layout requirements. It was developed in 1983 by the faculty and students at the University of California, Berkeley (UCB). Magic was primarily designed as an interactive color layout editing system for large-scale, custom design of MOS integrated circuits. Inherent within Magic are interactive packages that provide for easy modification, accurate designs, and convenient circuit extraction.

The Magic system has several key features, some of which are listed below:

1. Magic is a layout editor, thus it contains several user interactive layout editing operations.
2. Magic uses rectangular block or Manhattan style layouts.
3. It contains a background based incremental design rule checker which immediately identifies any layout design rule violations to the user. This checker has the capability to check both new and redesigned areas within the layout.
4. A hierarchical circuit extractor is used to convert the graphical layout into a file which contains information about the layout's environment, geometry, and connectivity. This file is used by the user to obtain simulation files.

The interested reader can find in-depth information about Magic within four articles describing the system [9][10][11][12].

B. SIMULATION FILES

The Magic extractor takes the layout information provided by the user and places it into a file that describes the circuit. This extracted file is given the filename extension "ext" and will be hereafter known as a .ext file. The .ext file contains information on the circuit's connectivity, transistor dimensions, as well as internodal capacitance and resistance. Basically, the circuit extractor defines the transistor nodes within a layout. This provides the capability for the layout to be functionally tested through simulation.

In order to perform simulation, a suitable simulation file must be produced. The program, "ext2sim", is a UCB tools program designed to flatten hierarchical .ext files. Flattening a file is a procedure which takes the geometry for a given area within the circuit and places it into a single set of tile planes. This procedure allows for the proper assignment of capacitances to overlapping cells [11][13]. Flattening a file places layout information for each transistor within a single record without explicitly providing hierarchical information. After flattening the file, "ext2sim" produces a simulation file that is suitable for use as input to simulators such as ESIM, or Crystal [13]. These simulators use the nodal information provided by the extractor and the "ext2sim" program to perform diagnostic tests for the user. Because it allows for the easy performance of simulations, the simulation file is an extremely valuable product of the Magic system.

The simulation file is given the filename extension "sim" by the "ext2sim" program. Within the remainder of this document, the simulation file will be denoted as a .sim file. Sample .sim files are discussed and illustrated in Chapter III.

[Note: Labels within the Magic file must be carefully selected. The convention of ending labels that are electrically connected throughout the circuit (such as Vdd and GND) with a "!" (e.g., Vdd! or GND!) must be addressed. This ending will

only appear within the Magic file. Once the file has been extracted and converted to a .sim file, the labels no longer contain the "!". Thus, it is extremely important for the designer to ensure that labels which are not electrically connected do not end with a "!". This will ensure that items with the same label which are not electrically connected can be distinguished.]

III. EXAMINING .SIM FILES OF KNOWN CMOS CIRCUITS.

There are several ways in which to represent a CMOS layout. A CMOS circuit diagram, a Magic layout, or a .sim file can all represent the same circuit. For example, the CMOS circuit illustrated in Figure 3.1 and the .sim file shown in Figure 3.2 are representations of the same inverter circuit. Consider now the manual recognition of a device (e.g., an inverter, passgate, etc.) within a given .sim file.

A. MANUAL RECOGNITION

To begin the task of recognizing all the elements within a CMOS layout, three layouts produced within Magic were considered: an inverter, a passgate, and a pseudo two-phase latch. The pseudo two-phase latch consists of two passgates and two inverters in series. While in Magic, these files were extracted using its hierarchical extractor. The extracted file was then converted to a UCB formatted .sim file using the "ext2sim" procedure. As mentioned in Chapter II, the .sim files produced are flat representations of each layout in a format that can be used during simulation.

The listings of these .sim files were studied to determine if the correct CMOS circuits could be manually reconstructed. One example listing is the inverter illustrated in Figure 3.2. This listing will be used as an initial reference point. The .sim files begin with a header line. This line begins with "|". The header line describes the scale factor, and the technology for the circuit. [Note: in some simulation files,

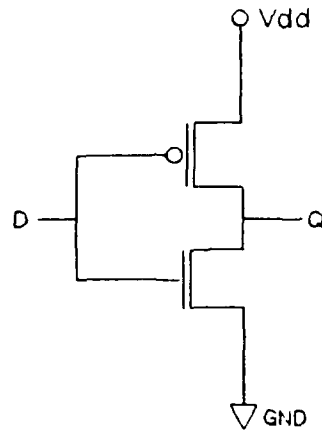


Figure 3.1: A CMOS inverter.

```

| units: 150 tech: scmos
p D Vdd Q 2 4 78 8
n D GND Q 2 4 78 -10
R GND 839
C Q GND 58
R Q 878
C D GND 13
R D 488
C Vdd GND 47
R Vdd 919

```

Figure 3.2: A .sim file for an inverter.

the header line also describes the format.] The header line is followed by the transistor lines. In Figure 3.2, the transistor lines are the second and third lines. These lines begin with a p or an n. The transistor lines are followed by the parasitic capacitance (if any is requested) and the parasitic resistance lines for each node. The capacitance lines always begin with a C, while the resistance lines begin with an R. The lines of primary concern within this thesis are those lines that describe the transistors.

```
p PHn D Q 2 4 78 8
n 3_337_14 3_270_404 3_2518_121 2 4 78 -10
```

Figure 3.3: Transistor lines in a .sim file.

Each transistor within a .sim file is described on a single line containing six or eight fields as illustrated in Figure 3.3. Each field is separated by a space. The first field is the transistor type. This field identifies the two different types of transistors, p-type and n-type. The .sim file distinguishes between the transistor types by using p to represent a p-type transistor and an n for the n-type transistor. The second field is the node representing the gate. The node has either a user defined labeled (e.g., PHn) or a Magic generated label (e.g., 3_337_14). In Figure 3.3, the p-type transistor contains all user defined labels, while the n-type transistor has Magic generated labels. The third and fourth fields are the transistor's source and drain respectively. Like the gate, the source and drain are either user labeled or Magic labeled. The fifth and sixth fields are the scaled length and width of the transistor. The final two fields (the seventh and eighth) are optional. If they are present, they describe the X and Y locations of a point inside the transistor.

The .sim file of the single inverter can be seen in Figure 3.2. Besides the header, the capacitance, and the resistance lines, the .sim file for an inverter contains two transistors. These transistors must be of different type (e.g., one transistor field is p and the other is n). The drain of the p-type must have the same label as the drain of the n-type. As seen in Figure 3.2, the drain of each transistor is labeled with a Q. The gates of the p-type and n-type must also have the same label (in this case D). The two sources must be labeled Vdd and GND.

The recognition of the CMOS inverter shown in Figure 3.1 can thus be accomplished by examining its .sim file (Figure 3.2). This recognition can be quickly

```

| units: 150 tech: scmos
p PHn D Q 3 4 98 30
n PH D Q 3 4 98 16
R GND 497
R PH 180
R Q 1238
R D 1238
R PHn 180
R Vdd 264

```

Figure 3.4: A .sim file for a passgate.

shown through the manual reconstruction of the inverter by examining its .sim file.

This reconstruction requires four steps:

1. Obtain a p-type and an n-type transistor.
2. If the drain or source of the p-type is the same as the drain or source of the n-type, then connect them.
3. If the gate of the p-type equals the gate of the n-type, then connect them.
4. If the nonmatching source or drain of one transistor (normally the p-type) is Vdd and the nonmatching source or drain of the other transistor (normally the n-type) is GND and steps 1 - 3 are true, then these two transistors make up one inverter.

The examination of the passgate's .sim file (illustrated in Figure 3.4) shows that its recognition follows a pattern similar to the inverter. The passgate also contains two transistors of differing types (e.g., p and n). As with the inverter, the drains must be labeled the same (in this case Q). The passgate, however, requires the two sources to be identical and the gates to be different but not Vdd or GND.

The reconstruction of the CMOS passgate (Figure 3.5) performed by examining its .sim file (Figure 3.4) also requires four steps:

1. Obtain a p-type and an n-type transistor.
2. If the drain or source of the p-type is the same as the drain or source of the n-type, then connect them.

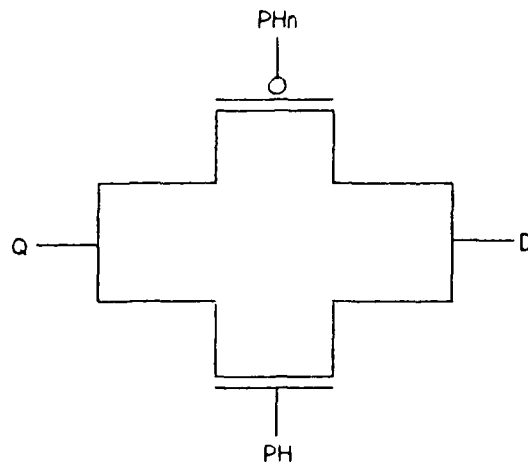


Figure 3.5: A CMOS passgate.

3. If the remaining drain(s) or source(s) of the p-type and n-type are equal, then connect them.
4. If neither gate is Vdd or GND and steps 1-3 are true, then these two transistors constitute a passgate.

Since it has been shown that it is possible to manually reconstruct one inverter and one passgate from their .sim listing, the next step is the implementation of an algorithm to automatically recognize these elements.

B. THE ALGORITHM

The reconstruction algorithm developed in this thesis project accomplishes three tasks. First, it accepts a .sim file as input. Next, it builds a linked list for the transistors. Finally, it searches the transistor list to obtain the proper matches for inverters and passgates.

The first step is to create dummy head and tail pointers for the linked lists. These pointers allow the quick recognition of the beginning and end of the linked lists. Separate linked lists are made for the transistors, the inverters and the passgates, because the basic algorithm is to be expanded later.

| units: 150 tech: scmos

Figure 3.6: A header line for a .sim file.

Next, the input file is read. This is accomplished one character at a time. A blank space or a new line separates each item of data. The header line (illustrated in Figure 3.6) must be checked first. The first item in the line must be a "|". If it is missing, then the input file is not a proper .sim file. The second item is the word "units:" which is followed by the scale factor (λ). All units (e.g., the transistors width, etc.) are multiplied by λ . The fourth item in the header line is the word "tech:" for technology. It is followed by the technology used within the .sim file. In some .sim files the header line also contains the format. This is not the case with the .sim file created with the 1986 version of the "ext2sim" program. If any of the header line data items are missing, the input file is not a proper .sim file, so the user need not continue.

If the header line is correct, the first transistor's information can be obtained from the next line. This line, like the header line, is read one character at a time. A blank space separates each data item (the transistor's fields). The first character in the line must be p or n. If the first character is a p or an n, a record (a data structure used to hold the transistor data) is created and placed at the end of the transistor's linked list. If the character is not a p or an n, then there are no more transistors within the .sim file.

Once the transistor's record has been created, the data line is read until the end of line marker. Each field is placed into its appropriate position within the record. When the end of line marker is reached, there is no more data for that transistor. The end of line marker constituting no more data is important because

the transistor's X and Y location fields are optional within UCB format. The reading of the input file character by character, line by line continues until the first character is not an n or p. A line not beginning with an n or p signals the end of the transistor listings in the .sim file. This completes the transistors' linked list. The total number of transistors in the file is recorded for historical purposes, so that the circuit designer can easily verify the total transistor count of the circuit.

The inverter list is built following the procedure outlined in the manual recognition section (Chapter III, Section A). This list is constructed by comparing the transistors, two at a time, to see if they make an inverter. The inverter algorithm follows:

As previously discussed, an inverter requires a p-type and an n-type transistor. Select the p-type and n-type transistors, compare their gates to see if they are the same. If they are the same, test for all the possible combinations for the other inverter connections. These combinations are:

1. The p-type source equals n-type source.
2. The p-type drain equals n-type source.
3. The p-type drain equals n-type drain.
4. The p-type source equals n-type drain.

If one of these combinations is found, test the nonmatching source (or drain) from both the p-type and n-type to determine if one is Vdd and the other is GND. If all these conditions are satisfied, then an inverter has been found. If one of these conditions are not satisfied, then these two transistors are not combined to make an inverter.

Once a inverter has been found, a record is established to hold both the inverters' transistors data. The record containing the data is placed at the end of the inverters' linked list. Comparing transistors continues until all the transistors have been compared. When the comparing is complete, so is the list of inverters in the circuit. [Note: this part of the algorithm was modified slightly to increase the speed of the comparison and will be discussed within the next section (Chapter III, Section C).]

Upon completing the inverter list, the passgate list is built following similar procedures. Two transistors are compared to determine if one is p-type and the other n-type, and that the gates are not Vdd or GND. Next, test for the two possible combinations for the other passgate connections. The possible combinations are:

1. The p-type source equals the n-type source and the p-type drain equals the n-type drain.
2. The p-type source equals the n-type drain and the p-type drain equals the n-type source.

If these connection requirements are met, then a passgate has been found. A record is established, filled with the transistors' data, and added at the end of the passgates linked list. The comparisons among the transistors continue until all have been compared.

The algorithm is essentially complete. Some administrative procedures can be added, such as recording the number of inverters and passgates or printing the lists of the transistors, passgates, and inverters as well as the header line information.

C. IMPLEMENTATION OF THE ALGORITHM

A program (see Appendix A) was written to implement the algorithm. The C programming language was used. Initially, the program was written so that the transistors were placed into a single transistor list. As the inverters and passgates were built, their transistors were not deleted from the list. This single list with no deletions was extremely time consuming when examining large circuits.

The C program was modified to improve this unacceptable search time. One modification was to delete the transistors after they had been used. This deletion is permissible because a transistor can only be used within a single component in the circuit. Since transistors are used to build passgates and inverters, inverters

and transistors are used to build NOR gates etc., transistor deletion will not effect future recognition algorithms because their information is still maintained within the record of the inverter or passgate. The deletion helped decrease the search time by reducing the number of comparisons.

Another modification placed each transistor into a list based on its type. Since two different types of transistors are required to build each device, placing the transistors into type lists prevents the comparison of a p-type transistor with a p-type transistor.

Still another modification to the algorithm was the combining of the inverter and passgate routines into one function. This new function searched the transistor lists for inverters and passgates at the same time. This same time search allowed the comparison of the same two transistors to be performed in a single search through the transistor lists rather than two searches. The modification eliminated the duplicate search of the transistor lists which was performed in the initial search scheme. This elimination decreased the search time.

Table 3.1 illustrates the time reductions achieved with each of these modifications. The table's information was obtained by using the VAX 11/785 system "time" and running each program. The input file used was a sixteen bit ALU that contains a total of 1632 transistors, 144 inverters, and 192 passgates. The user time (the time the program spent in the system), the system time (the time spent executing the command), and the elapse time (the total time required to executed the command) provide the most important information.

In examining Table 3.1, the effectiveness of the deletion program comes into question, because its time increases in two categories. The slight increases (9% in user time and 8% in elapse time) are offset by the 29% decrease in system time. The deletion program is also the catalyst for the overall reduction obtained with the

TABLE 3.1: Comparisons of Program Speeds on the VAX 11/785

| TECHNIQUES | USER TIME (sec) | SYSTEM TIME (sec) | ELAPSE TIME (min:sec) | PERCENT CPU CYCLE USED | AVERAGE MEMORY/DATA USAGE | NUMBER DISC READS WRITES | NUMBER PAGE FAULTS SWAPS |
|------------------|-----------------------|-------------------------|-----------------------------|------------------------------|---------------------------------|-----------------------------------|-----------------------------------|
| INITIAL PROGRAM | 76.1u | 1.4s | 2:34 | 50% | 21+750k | 11+44in | 0pf+0w |
| DELETION PROGRAM | 83.3u | 1.0s | 2:47 | 50% | 23+500k | 13+23in | 0pf+0w |
| TWO LINKED LISTS | 50.5u | 0.8s | 1:43 | 49% | 25+576k | 13+24in | 0pf+0w |
| SAME TIME SEARCH | 43.7u | 0.9s | 1:35 | 46% | 25+578k | 12+23in | 0pf+0w |

TABLE 3.2: Comparisons of Program Speeds on ISI

| TECHNIQUES | USER TIME (sec) | SYSTEM TIME (sec) | ELAPSE TIME (min:sec) | PERCENT CPU CYCLE USED | AVERAGE MEMORY/DATA USAGE | NUMBER DISC READS WRITES | NUMBER PAGE FAULTS SWAPS |
|------------------|-----------------------|-------------------------|-----------------------------|------------------------------|---------------------------------|-----------------------------------|-----------------------------------|
| INITIAL PROGRAM | 32.0u | 0.6s | 0:33 | 98% | 1+91k | 26+64in | 4pf+0w |
| DELETION PROGRAM | 28.1u | 0.6s | 0:30 | 95% | 2+83k | 25+41in | 5pf+0w |
| TWO LINKED LISTS | 18.5u | 0.5s | 0:24 | 77% | 2+83k | 24+43in | 5pf+0w |
| SAME TIME SEARCH | 14.3u | 0.4s | 0:15 | 96% | 2+82k | 26+40in | 5pf+0w |

modifications between the initial program and the same time search. The overall reduction in user time is 48%, system time is 43%, and elapse time is 43%.

To further demonstrate the significant reduction in the program's runtime, the same programs with the common input file were run on an Integrated Solution Optimum V (ISI) workstation. This workstation is equipped with a Color VME-Graphics subsystem. The ISI has four megabytes of memory and approximately one gigabyte of disk storage. It contains a high speed MC6881 Floating-Point Co-processor. Its CPU is a 16.67 MHz MC68020. Table 3.2 illustrates the significant timing reductions received with each modification. Once again the overall reductions are large (56%, 33%, and 55% respectively). There was even a decrease in all time between the initial program and the deletion program. This decrease can be attributed to the lack of a system load on the ISI workstation.

Both Table 3.1 and Table 3.2 indicate that the initial program's runtime is excessively high. They also demonstrate that the unacceptable search time was improved significantly after the third modification was implemented. Because of this improvement, the third modification (the same time search routine) will be utilized in the next stages of research.

D. ALGORITHM ACCOMPLISHMENTS

Transistors are the initial (level0) elements in the recognition process. They are identified directly from the .sim file. The recognition of level0 components (transistors) and the building of the transistor list allows for the comparison of transistors to find and build a new level of circuit components. The passgates and inverters are the initial elements of this new level of components. It is called level1.

The inverter and passgate algorithms lead to the next phase of research. It has been shown that transistors, passgates, and inverters can be built from a .sim file. The next phase will focus on the joining of the remaining transistors into other level1 components and the identification of higher level components.

IV. DEVELOPING CONNECTIVITY TO MORE TRANSISTORS.

A. LEVEL1 MANUAL RECOGNITION

Two level1 components have already been identified, the inverter and the passgate. The remaining level1 elements were divided into two categories:

1. Groups of transistors with at least one connection to Vdd and the remaining transistors in the group providing a path to GND. This category provides numerous connection possibilities, because the connections can be made both serially and in parallel. Two sample groupings are illustrated in Figure 4.1.
2. Groups of transistors which have no immediate connection to Vdd. This category is found after all of the other level1 devices have been identified. Like the previous category, it also has numerous connection possibilities. Most noteworthy are the connections which form multiplexers, encoders, and decoders. Figure 4.2 illustrates a sample category 2 device.

These final two level1 categories use the transistors within the .sim file that were not identified as part of an inverter or passgate. Therefore, the remaining .sim transistors are examined to recognize these two categories.

The manual recognition of the level1 element that has a connectivity to Vdd will be addressed first. The process of reconstructing this level1 element requires four steps. First, a transistor connected to Vdd is selected from the remaining .sim transistors. Next, all the transistors that are connected to the source or drain of the initially selected transistor are found. This connection cannot be through Vdd because Vdd is normally common throughout a CMOS circuit. Connecting all elements that are connected to Vdd would defeat the purpose of this level1 recognition, because every group with a connection to Vdd would be placed within a single device. Figure 4.3(a) is a proper level1 connection, while (b) is not. (Figure 4.3(b)

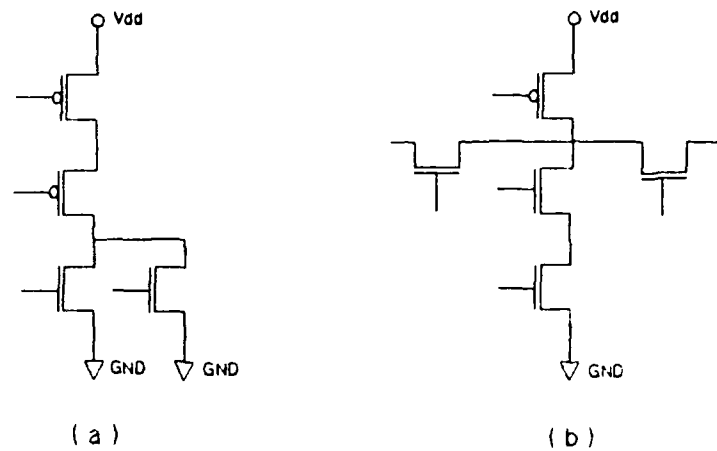


Figure 4.1: Two Level1 Category 1 Devices

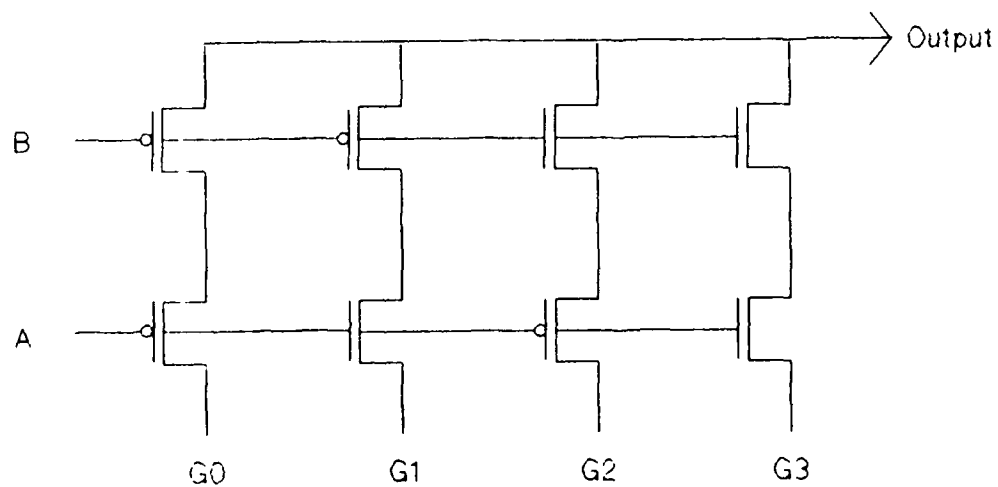


Figure 4.2: A Level1 Category 2 Device

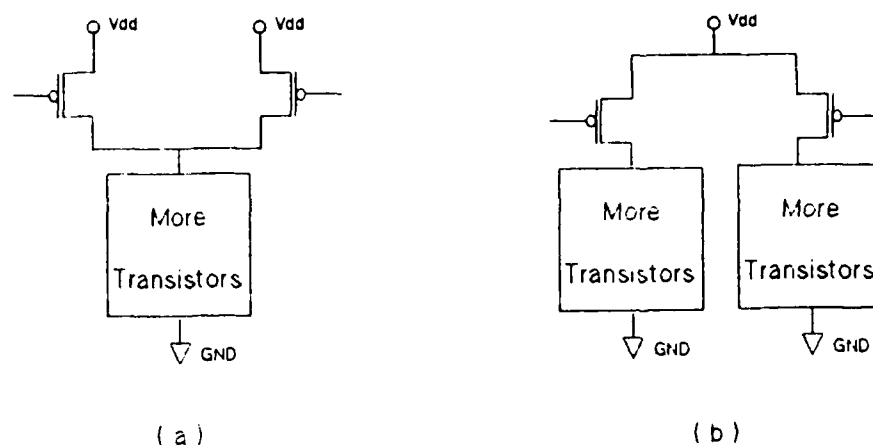


Figure 4.3: Connectivity with Vdd

contains two level1 connections). The identification of all the transistors that can be connected to the newly found transistors is continued until all are found. After finding all of the connecting transistors, determine whether or not any of these newly found transistors are connected to GND. If one is connected to GND, then the process of reconstructing this element is complete. If none of these transistors are connected to GND (see Figure 4.4), then there is a possible error in this .sim file because a connection to Vdd exists with no path to GND. This process is repeated until there are no transistors remaining with connectivity to Vdd within the .sim file. When no transistors connected to Vdd remain, the manual reconstruction of all the first category of level1 elements is complete.

In order to manually recognize the second category of level1 elements, all the first category elements must have already been identified. This is due to the second category reconstruction process requirement of every transistor remaining within the .sim file being connected to every other transistor within the .sim file whose source or drain matches. This connectivity provides for the proper recognition of devices such as the encoder in Figure 4.2. Thus the manual reconstruction

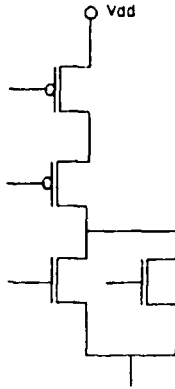


Figure 4.4: No Connectivity with GND

of the second category of level1 elements is a repetitious process which contains one or more steps. The first step is to select a transistor in the .sim file. Then determine whether any other transistor will connect to the selected transistor's source or drain. If a connecting transistor is found, the process is repeated to see if any other transistors can be connected to any of the newly found transistors. This connecting continues until no more transistors can be found whose drain or source match the drain or source of any of the second category components. If none are found, the process of recognizing this second category element is complete.

It should be noted that the second category elements may contain just one transistor. This will occur if no other transistors that remain in the .sim file can be connected to the initially selected transistor's source or drain. This is important because a single transistor will provide a quick flag to the designer, if a stray transistor was not purposely designed within the circuit. Furthermore, allowing a single transistor to reside within a level1 device eliminates the need for level0 searches within higher level recognition algorithms.

B. LEVEL2 MANUAL RECOGNITION

Once all the level1 devices have been identified, level2 recognition can begin. The level2 components are identified by connecting level1 components together. With this in mind, the manual recognition of a level2 device is relatively straightforward.

A level1 device is selected as the first component of a level2 device. This selection is random; choosing a different level1 device as the first component will not effect the construction of the level2 device. All the remaining level1 devices are examined to determine whether or not any can be connected to the first level2 component. If one or more level1 devices can be connected, the remaining level1 devices are examined to see if any can be connected to these newly found level2 components. These examinations continue until no more level1 devices can be connected to any of the level2 components. At this point, a level2 device has been identified. The level2 manual recognition process is repeated until no level1 devices remain. No remaining level1 devices signifies that all the level2 devices have been found.

Ideally a .sim file will contain a single level2 device. This level2 device will represent the original CMOS layout. If the layout was a properly constructed circuit, only one level2 device will appear. However, if the CMOS layout contains representations of two or more distinct circuits then two or more level2 devices will be identified.

C. LEVEL1 ALGORITHM

The third modification to the algorithm developed in the initial research phase (see Chapter III, Section C) placed the transistors into two separate lists. That

version of the recognition algorithm was built upon in order to provide further level1 recognition.

As mentioned in the level1 manual recognition section (Chapter IV, Section A), level1 recognition was divided into two categories. The first category contains those level1 devices that are connected to Vdd. The second category are the remaining level1 devices (those that are not connected to Vdd).

Identifying all the level1 devices that are connected to Vdd requires the following eight step algorithm:

1. Search the p-type transistor list for a transistor whose gate, source, or drain is Vdd. If one is found, remove it from the linked list. This will prevent this transistor from being selected again.
2. Search the p-type transistor list for any transistors whose source or drain are connected to the source or drain of the transistor found in step 1. If a p-type transistor is found, then remove it from the list.
3. Search the n-type transistor list to find all the transistors whose drain or source are connected to the drain or source of the p-type transistors found in steps 1 and 2. Remove all these matching transistors from the n-type list.
4. Search the p-type transistor list to find all the transistors whose drain or source are connected to the drain or source of the n-type transistors found in step 3. Remove all these matching transistors from the p-type list.
5. Search the n-type transistor list to find all the transistors whose drain or source are connected to the drain or source of the p-type transistors found in step 4. Remove all these matching transistors from the n-type list.
6. Repeat steps 4 and 5 until no connecting transistors remain in the p-type or n-type transistor lists.
7. Check all the n-type transistors found in step 3 to determine whether or not any has a gate, source, or drain which is GND. If no n-type transistor is connected to GND, then an error may have occurred, because there is no path from Vdd to GND.
8. All the transistors that were identified in steps 1-7 constitute a level1 device and are placed into an array for that particular device. Once the array is built, the device counter (which identifies the array) is incremented.

The eight steps above are repeated until there are no more p-type transistors which are connected to Vdd within the p-type transistor list. When no more p-type transistors that have Vdd as the gate, source, or drain remain on the list, this category of level1 devices is complete.

As mentioned in step 7, there is a possibility that no transistor that is connected to GND can be found to connect to this level1 device. This could happen, for instance, if the designer wants particular transistors to be constantly high. The implications of this situation will be discussed more in depth within the implementation section (Chapter VI, Section E).

The second category of the level1 algorithm requires all the remaining transistors which can be connected to be placed into a level1 device. Identifying these level1 elements is accomplished using the following six step algorithm:

1. Combine the p-type and n-type transistor lists into a single list. [Note: this combining makes the mechanics of the recursive algorithm easier to implement, especially since the type of transistor is no longer a concern as it was with the passgate and inverter.]
2. Select the first transistor on the newly combined list. Place it on top of a stack and remove it from the list.
3. Compare the drain and the source of all the transistors on the list with the drain and source of the transistor on the top of the stack. Once a match is found, place the matching transistor on the top of the stack and remove it from the list.
4. Continue step 3 until no transistor can be found with a matching source or drain. Remember, always compare the list transistors to the transistor on the top of the stack. Once no matching transistors are found, place the transistor on the top of the stack into the device array and decrement the stack.
5. Continue steps 3 and 4 until there are no transistors remaining on the stack. The resulting device array of transistors constitutes a complete level1 device.
6. Increment the device number and repeat step 2-6 until no transistors remain on the newly combined list.

When no transistors remain on the combined transistor list, all the level1 devices have been found. Thus the entire level1 algorithm is concluded.

D. LEVEL2 ALGORITHM

The level2 algorithm requires the comparison of all the level1 devices. The algorithm requires the following three steps:

1. Select one level1 device and designate it as a level2 device.
2. Select another level1 device. Compare the gate, source, and drain of every transistor within the level1 device with the gate, source, and drain of every transistor within the level2 device. If a match occurs, then add this level1 device to the level2 device. If no match occurs, then continue.
3. Repeat step 2 until no more level1 devices can be found whose transistors can connect to the level2 device.

When no more level1 devices remain, the level2 algorithm is finished.

E. IMPLEMENTATION OF HIGHER LEVEL ALGORITHMS

The higher level algorithms (level1 and level2) were written as procedures and were added to the C program previously addressed in Chapter III. These algorithms which were discussed in the previous two sections have been implemented. During their implementation, slight modifications to existing data structures and algorithms were necessary. These modifications are described herein.

Developing a standardized data structure for level1 devices would require prior knowledge of each circuit being examined by this recognition program. Since this prior knowledge is not always possible and more importantly because the program is responsible for recognizing what is in the circuit, a standardized data structure is not possible. To allow for flexibility and enable the program to perform future searches, a two-dimensional array was used to house level1 device elements (e.g., `level1[numdevice][numtrans]`). The first field in the array (`numdevice`) contains the device number, while the second field (`numtrans`) contains pointers to each transistor that makes up the device. Using arrays could cause the program to be redimensioned prior to execution because the array size must be declared within

the program. The redimensioning would be necessary especially if memory space is limited and a large circuit is being analyzed. However, redimensioning is far easier than revising a standardized level1 data structure each time the tested circuit is changed.

Initially, the level1 algorithm requires Vdd to have a path to GND. This requirement may not always be the case within a circuit. The designer may have transistors which are constantly high or low for generating constants. To accommodate a Vdd with no path to GND, an interactive error routine was established. The routine now asks the user if he or she wishes to reexamine the .sim file because the circuit was designed with no path to GND. If the user answers no, the program would allow this device to be entered as a level1 device and would continue execution. If the user answer was yes, then an error message would appear followed by the termination of the program.

The algorithm for the second category of level1 elements (those not connected to Vdd) was implemented using a recursive procedure. After linking the two transistor lists together, this procedure recursively calls itself until all of the possible transistor connections for that particular level1 device have been found. This implementation of the recursive algorithm provides the proper identification, while requiring less written code than a nonrecursive algorithm.

The level2 device structure presented a dilemma. The standardized data structure of the inverter and passgate either had to be changed or a different data structure used. Rather than changing the data structure, three two-dimensional arrays as well as one three-dimensional array were studied as possible level2 data structures. There are advantages for selecting either structure. The three-dimensional array would require approximately the same memory space as the three two-dimensional arrays, because the same information would have to be stored no matter which

data structure was selected. The three-dimensional array would be less cumbersome in keeping track of the data since three separate arrays would not have to be accessed. Unfortunately, the three-dimensional array structure is extremely difficult to implement in the C programming language because of the three differing data types (transistors, passgates, and inverters). Thus three two-dimensional arrays were used to house the level2 elements. One containing each of the level1 transistor devices, inverter devices, and passgate devices. This structure provides for the easy addition of other devices. As with the level1 transistor device, the first field of the array contains the level2 device number, while the second field points to one of the level1 elements.

Step 2 of the three step level2 algorithm was implemented by repeating the following nine comparisons until no more matches were found.

1. Compare level1 transistor devices with level2 transistor arrays.
2. Compare level1 passgate devices with level2 transistor arrays.
3. Compare level1 inverter devices with level2 transistor arrays.
4. Compare level1 transistor devices with level2 inverter arrays.
5. Compare level1 passgate devices with level2 inverter arrays.
6. Compare level1 inverter devices with level2 inverter arrays.
7. Compare level1 transistor devices with level2 passgate arrays.
8. Compare level1 passgate devices with level2 passgate arrays.
9. Compare level1 inverter devices with level2 passgate arrays.

The implementation of the above modifications provided the recognition program with more flexibility and versatility. They also allow the easy addition of new devices to be performed. Furthermore, the modifications require less knowledge by the user to utilize the program, while providing accurate results.

F. HIGHER LEVEL ALGORITHM ACCOMPLISHMENTS

Using a .sim file of a properly constructed circuit, the recognition algorithm will identify only one level2 device. The algorithm's transistor, passgate, and inverter count will be identical to those found in the level1 devices. If a .sim file has been constructed with two distinct circuits, then the level2 algorithm will produce two level2 devices. Since a normal .sim file contains only one circuit, identifying .sim files with more than one circuit is a very beneficial feature of the level2 algorithm.

The recognition algorithm was used to identify minor changes within the sixteen bit ALU describe in Chapter III, Section C. The ALU was modified slightly by first removing a transistor from four bit slices. The modified ALU was tested by the recognition algorithm. By comparing the results of the correct version's output to the results of the modified version's output, the identities of the removed transistors were found.

The ALU was again modified by removing a transistor which provided a path from Vdd to GND. The missing path to GND was quickly identified by the algorithm. Then by again comparing the modified version's output with the correct version's output, the identity of the missing transistor was determined.

V. CONCLUSIONS AND RECOMMENDATIONS.

A. THE RECOGNITION ALGORITHM

The recognition algorithm, provided within Appendix A, achieves the primary goal of the thesis. That goal is to develop an algorithm to recognize different elements within the CMOS circuit given only the .sim file for that circuit. The goal was met by first developing an algorithm which would properly identify inverters and passgates. The algorithm was improved by expanding it to recognize other level1 devices, and then recombining the circuit into a level2 device.

The algorithm's validation process reaffirmed the successful accomplishment of the primary goal. Numerous sample .sim files were developed and successfully analyzed by the recognition program. One such sample file was the sixteen bit ALU which was utilized in the verification process as discussed in Chapter III, Section C and Chapter IV, Section F. Other sample .sim files include the Corn88 chip [14], modified bit slices of the sixteen bit ALU, a pseudo two-phase latch, as well as specifically designed files to test the features of each algorithm.

B. FUTURE RESEARCH

As VLSI circuits become more complex, the ability to verify circuit design and timing becomes increasingly important. As a result of this research, a better understanding of the problems involved in the quick verification of VLSI circuits has been achieved.

The recognition algorithm presented here is the first step in verifying the actual VLSI chip design. It would be used to provide the initial mechanism which

when built upon will give designers the capability to quickly and accurately verify a VLSI circuit's design and timing. Here is how it may work.

The recognition algorithm would be used to locate, isolate, classify, and count the level1 devices. The data provided by the algorithm would be compared to the original specifications. This comparison would allow the designers to verify the design or identify faults. This is accomplished by verifying that the correct number of level1 devices (e.g., inverters) are contained within the diagram. Furthermore, the location of each device within the circuit could also be verified by comparing the actual location with the design specifications. The recognition algorithm is a verification tool not a correction device. For example, if a designer inadvertently places an additional transistor in an inverter, the algorithm will count these three transistors and identify one inverter among the other components in the circuit. By reviewing and correctly interpreting the recognition algorithm's output data, the designer can identify his or her mistakes.

Upon completing the design verification, the data could be used to perform timing analysis through a table lookup scheme. Since the devices have been classified, a table for device-level delays could be developed. Then by using this table of device delay times along with connectivity requirements, circuit timing analysis could be performed. This analysis could reduce the time required for the verification process because device-level timing would be performed rather than node-level timing.

Utilizing the algorithm as the initial stage of a much larger VLSI design verifier provides numerous research possibilities. Some noteworthy topics include the circuit simulator, the timing analyzer, and user interface requirements.

APPENDIX A: PROGRAM LISTINGS

A. GLOBAL VARIABLES

```

/*****
** JOEL V. SWISHER          Thesis #1          EC3820          **
** This program contains the common definitions used in all    **
** thesis #1 input files.                                     **
*****/
#include <string.h>
#include <malloc.h>
#include <stdio.h>
typedef long Boolean;
/*****
** Macro definitions.                                          **
*****/
#define TRUE 1
#define FALSE (!TRUE)
#define MAXLEN 33
#define MALLOC(x) ((x *) malloc(sizeof(x)))
#define IsWhite(x) ((x=='\n') || (x==' ') || (x=='\t'))
#define UNITLEN 4
#define TECHLEN 5
#define IsDigit(x) ((060 <= x) && (x<=071)) /*gives boundaries */
#define newline(x) ((x=='\n'))

/*****
** Devicenode contains a pointer to the transistor type; plus **
** pointers to the source, drain, and gate of the transistor; **
** It contains information on the transistor's width, length, **
** and location; and finally, a pointer to the next transistor **
** in the linked list.                                       **
*****/
struct devicenode
{
char    *Type;
char    *gate;
char    *source;
char    *drain;
struct devicenode *next;
char    *length;
char    *width;

```

```

char    *xloc;
char    *yloc;
char    *use_lv2;
};
typedef struct devicenode trans; /* define trans as type struct devicenode */

/*****
** Invertnode contains pointers to the common gate and drain; **
** as well as the information on the n-type and p-type **
** transistors which make up the inverter; and finally a link **
** to the next inverter in the list. **
*****/
struct invertnode
{
char    *gate;
char    *source_ptype;
char    *source_ntype;
char    *drain_ptype;
char    *drain_ntype;
char    *length_ptype;
char    *length_ntype;
char    *width_ptype;
char    *width_ntype;
char    *xloc_ptype;
char    *xloc_ntype;
char    *yloc_ptype;
char    *yloc_ntype;
struct invertnode *next;
};
typedef struct invertnode inv; /* define inv as type struct invertnode */

/*****
** Passgatenode contains pointers to the common source and **
** drain; as well as the information on the n-type and p-type **
** transistors which make up the passgate; and finally a link **
** to the next passgate in the list. Note: the to sources are **
** needed to determine which nodes (source/source or **
** source/drain) are connected. **
*****/
struct passnode
{
char    *gate_ptype;
char    *gate_ntype;
char    *source_ptype;
char    *source_ntype;

```

```

char    *drain;
        char    *length_ptype;
        char    *length_ntype;
char    *width_ptype;
char    *width_ntype;
char    *xloc_ptype;
char    *xloc_ntype;
char    *yloc_ptype;
char    *yloc_ntype;
struct  passnode *next;
};
typedef struct passnode pass; /* define pass as type struct passnode */

/*****
** header contains the length, a head ptr, and a tail ptr.      **
*****/
struct header
{
    int          length;
    trans        *head, *tail;
};
typedef struct header head_type;
head_type      *header_newp; /* header_new is of type struct header */
head_type      *header_newn; /* header_new is of type struct header */

/*****
** headinv contains the length, a head ptr, and a tail ptr.     **
*****/
struct headinv
{
    int          length;
    inv          *head, *tail;
};
typedef struct headinv head_inv;
head_inv       *head_invert; /* head_invert is of type struct head_inv */

/*****
** headpass contains the length, a head ptr, and a tail ptr.    **
*****/
struct headpass
{
    int          length;
    pass         *head, *tail;
};
typedef struct headpass head_pass;

```

```

head_pass      *head_passgate; /* head_passgate is of type struct head_pass
*/

/*****
** External Functions
*****/
extern  filler();
extern  trans *NewNode();
extern  inv  *NewInvert();
extern  pass *NewPass();
extern  print_stats1();
extern  print_stats2();
extern  print_ptrans();
extern  print_ntrans();
extern  print_invert();
extern  print_pass();
extern  head_type *create();
extern  head_inv  *createinv();
extern  head_pass *createpass();
extern  ptransistor();
extern  ntransistor();
extern  invpass();

/*****
** Global Variables
*****/
FILE  *fo;      /* fo is a pointer to the output file */
FILE  *fp;      /* fp is a pointer to FILE */
extern FILE  *fp; /* fp is a pointer to FILE */
trans *newp;
trans *newn;
inv   *newinv;
pass  *newpass;
char  scale[UNITLEN]; /* scale is the char field in the .sim file which
                        when multiplied times all of the diminsions will
                        will give centrimicrons. */
char  technology[TECHLEN]; /* The technology used in the VLSI circuit */
char  buffer[MAXLEN+1]; /* data holding place */
char  buf[MAXLEN+1]; /* data holding place */
int    Check;
int    scalefactor; /* A decimal conversion of the scale */
int    len; /* length used in filler() */
int    complete; /* at EOF */
int    total_transistors;
int    total_passgates;

```

```

int total_invert;

/*****
** Thesis rec.h items for level1 and level2.      **
*****/

#define NUMDEVICE 495
#define NUMTRANS 2690
head_type *header_new; /* header_new is of type struct header */

/*****
** level1 is a two-dimensional array of pointers to trans.      **
*****/
trans *(level1[NUMDEVICE] [NUMTRANS]);

/*****
** stack is a one-dimensional array of pointers to trans.      **
*****/
trans *(stack[NUMTRANS]);

/*****
** level2 is a two-dimensional array of pointers to trans.      **
** plevel2 is a two-dimensional array of pointers to passgates. **
** ilevel2 is a two-dimensional array of pointers to inverters. **
*****/
trans *(level2[NUMDEVICE] [NUMTRANS]);
pass *(plevel2[NUMDEVICE] [NUMTRANS]);
inv *(ilevel2[NUMDEVICE] [NUMTRANS]);

extern print_level1();
extern Pop();
extern Push();
extern combine();
extern combinep();
extern combinen();
extern checkp();
extern fil_tran2();
extern fil_pass2();
extern fil_inv2();
extern error_level2();

/*****
** pcount and ncount contain the number of p-type or ntype      **
** transistors per level1 device.      **
*****/

```

```
int stacknum;
int pcount[NUMDEVICE];
int ncount[NUMDEVICE];
int tcount[NUMDEVICE];
int numn,nump,numdevice,numtrans; /* counters */
int ground;
int nolvl;
int vice;
int lv2;
int tent2[NUMDEVICE];
int icnt2[NUMDEVICE];
int pcnt2[NUMDEVICE];
```

B. MAIN PROGRAM

```

/*****
** JOEL V. SWISHER          Thesis #1
** This program reads a.sim file as input. Then searches that
** file for possible passgates or inverters
*****/

#include "rec.h"
main(argc,argv) /* argc and argv communicate with the operating system */
int argc;      /* argc provides a count of the number of command line
                arguments */
char **argv;   /* argv is an array of pointers to char */
{
    if (argc == 1) /*Is the command line entry, entered properly? */
    {
        printf("Please enter a data file name after the prog name.\n");
        exit(-1); /* if not, print then exit */
    }
    /*****
    ** Create the initial head and tail pointers for the linked lists. **
    *****/
    header_newp = create();
    header_newn = create();
    head_invert = createinv();
    head_passgate = createpass();
    fp = fopen(argv[1], "r"); /* fopen opens argv[1] for reading */
    fo = fopen("out.rlv2", "w"); /* opens out for writing */
    len = 0;
    /*****
    ** Call filler() to obtain the first character in the file.      **
    *****/
    filler();
    if((strcmp(buffer,"|"))!=0)
        error();
    /*****
    ** Is this a proper simulation file? If so, continue.          **
    *****/
    else
    {
        filler();
        if((strcmp(buffer,"units:"))==0)
        /*****
        ** Is this still a proper simulation file?                  **
        ** If so, determine scalefactor.                            **
        *****/

```



```

{
    filler();
    strcpy(scale,buffer);
}

else error();
    filler();
    if((strcmp(buffer,"tech:")!=0)
/*****
** Is this still a proper simulation file?          **
** If so, determine the technology used.             **
*****/
{
    filler();
    strcpy(technology,buffer);
}

else error();
    filler();
/*****
** Since this is a proper simulation file (to this point), fill **
** the transistor list. The next several lines of data from the **
** input file contain information on each transistor.          **
*****/
total_transistors=0;
while(((strcmp(buffer,"n")!=0)
||((strcmp(buffer,"p")!=0))
    {
/*****
** A call is made to Transistor() which builds the transistor **
** list. Upon exiting transistor() obtain the first character **
** in the next line, then continue the while if required.      **
*****/
if((strcmp(buffer,"p")!=0)
ptransistor();
else ntransistor();
total_transistors++;
    }

/*****
** Build the inverter list.                                     **
** Build the passgate list.                                     **
*****/
{
    invpass();
    total_invert=head_invert->length;
    total_passgates=head_passgate->length;
}

```

```

    {
/*****
** Build the remaining level1 device array of pointers to trans.**
*****/
level_one();
/*****
** Must print the statistics and information on transistors      **
** inverters and passgates prior to entering level2 recognition.**
*****/
fprintf(fo,"no more transistors.\n");
print_stats1();
    }
    {
/*****
** Build the remaining level2 device array of pointers to trans,**
** inv. and pass.                                               **
*****/
level_two();
    }
}
print_ptrans();
print_ntrans();
print_level1();
print_stats2();
fclose(fp);
fclose(fo);
exit(0);
}

```

C. TRANSISTOR LIST FUNCTION

```

/*****
** JOEL V. SWISHER          Trans.c          **
** This function builds the transisitor and places it in the **
** linked list.          **
*****/

#include "rec.h"
ptransistor()
{
    trans    *curr;
    int      done;
    curr=header_newp->tail;
    done=FALSE;
    Check=FALSE;
/*****
** Increment header_newp->length which is the length for **
** printing.          **
*****/
    header_newp->length++;
    while(!done)
    {
/*****
** Create a node in the link list to hold the transistor **
** information.          **
*****/
        newp = NewNode();
        newp->Type=malloc(sizeof(char)*strlen(buffer+1));
        strcpy(newp->Type,buffer);
/*****
** Is this the first entry on the list? If not, this is the **
** last entry.          **
*****/
        if (header_newp->head==NULL)
        {
            header_newp->head=newp;
            header_newp->tail=newp;
        }
        else
        {
            header_newp->tail=newp;
            curr->next=newp;
        }
/*****

```

```

** Within a sim file, each transistor's information is      **
** contained on a single line. Successive calls are made to  **
** filler() to get the next information field. Since xloc and **
** yloc are optional fields Check is see in filler at the end **
** of a line.                                               **
*****/
    while(!(Check))
    {
        filler();
        newp->gate=malloc(sizeof(char)*strlen(buffer)+1);
        strcpy(newp->gate,buffer);
        filler();
        newp->source=malloc(sizeof(char)*strlen(buffer)+1);
        strcpy(newp->source,buffer);
        filler();
        newp->drain=malloc(sizeof(char)*strlen(buffer)+1);
        strcpy(newp->drain,buffer);
        filler();
        newp->length=malloc(sizeof(char)*strlen(buffer)+1);
        strcpy(newp->length,buffer);
        filler();
        newp->width=malloc(sizeof(char)*strlen(buffer)+1);
        strcpy(newp->width,buffer);
        filler();
        newp->xloc=malloc(sizeof(char)*strlen(buffer)+1);
        strcpy(newp->xloc,buffer);
        filler();
        newp->yloc=malloc(sizeof(char)*strlen(buffer)+1);
        strcpy(newp->yloc,buffer);
    }
    done=TRUE;
}

/*****
** This assists the main program in preparing for the next    **
** iteration of this function call by getting the first      **
** character in the next line on the sim file.               **
*****/
filler();
}

/*****
** This function builds the transistor and places it in the   **
** linked list.                                               **
*****/
ntransistor()
{

```

```

trans  *curr;
int     done;
    curr=header_newn->tail;
    done=FALSE;
    Check=FALSE;
/*****
** Increment header_newn->length which is the length for      **
** printing.                                                  **
*****/
    header_newn->length++;
    while(!done)
    {
/*****
** Create a node in the link list to hold the transistor      **
** information.                                                **
*****/
        newn = NewNode();
        newn->Type=malloc(sizeof(char)*strlen(buffer+1));
        strcpy(newn->Type,buffer);
/*****
** Is this the first entry on the list? If not, this is the  **
** last entry.                                                **
*****/
        if (header_newn->head==NULL)
        {
            header_newn->head=newn;
            header_newn->tail=newn;
        }
        else
        {
            header_newn->tail=newn;
            curr->next=newn;
        }
/*****
** Within a sim file, each transistor's information is      **
** contained on a single line. Successive calls are made to  **
** filler() to get the next information field. Since xloc and **
** yloc are optional fields Check is see in filler at the end **
** of a line.                                                **
*****/
        while(!(Check))
        {
            filler();
            newn->gate=malloc(sizeof(char)*strlen(buffer)+1);
            strcpy(newn->gate,buffer);

```

```

filler();
newn->source=malloc(sizeof(char)*strlen(buffer)+1);
strcpy(newn->source,buffer);
filler();
newn->drain=malloc(sizeof(char)*strlen(buffer)+1);
strcpy(newn->drain,buffer);
filler();
newn->length=malloc(sizeof(char)*strlen(buffer)+1);
strcpy(newn->length,buffer);
filler();
newn->width=malloc(sizeof(char)*strlen(buffer)+1);
strcpy(newn->width,buffer);
filler();
newn->xloc=malloc(sizeof(char)*strlen(buffer)+1);
strcpy(newn->xloc,buffer);
filler();
newn->yloc=malloc(sizeof(char)*strlen(buffer)+1);
strcpy(newn->yloc,buffer);
}
    done=TRUE;
}
/*****
** This assists the main program in preparing for the next
** iteration of this function call by getting the first
** character in the next line on the sim file.
*****/
filler();
}

```

D. INVERTER AND PASSGATE RECOGNITION FUNCTION

```

/*****
** JOEL V. SWISHER          Inverter.c          **
** This function builds the inverters and places it in the **
** linked list.          **
*****/

#include "rec.h"
invpass()
{
    int    done;
    int    complete;
    int    stop,found;
    trans  *second, *first;
    trans  *prev1st, *prev2nd;
    pass    *curr1;
    inv     *curr;

    first=header_newp->head;
    second=header_newn->head;
    prev1st=first;
    prev2nd=second;

/*****
** Go through the transistor list to find an inverter or **
** passgate.          **
*****/
    stop=FALSE;
    while((first!=NULL)&&(stop==FALSE))
    {
        complete=FALSE;
        done=FALSE;
        while((second!=NULL)&&(!done))
        {
            found=FALSE;
/*****
** Is it an inverter?          **
*****/
            if(((strcmp(second->gate,first->gate))==0)&&
                ((strcmp(second->drain,first->drain))==0)&&
                (((strcmp(second->source,"Vdd"))==0)||
                 ((strcmp(first->source,"Vdd"))==0))&&
                (((strcmp(second->source,"GND"))==0)||
                 ((strcmp(first->source,"GND"))==0))&&
                ((strcmp(first->source,second->source))!=0)&&
                ((strcmp(second->Type,first->Type))!=0))

```

```

{
    found=TRUE;
}
else if(((strcmp(second->gate,first->gate))==0)&&
        ((strcmp(second->source,first->source))==0)&&
        (((strcmp(second->drain,"Vdd")==0)||
          (strcmp(first->drain,"Vdd")==0))&&
          ((strcmp(second->drain,"GND")==0)||
            (strcmp(first->drain,"GND")==0))&&
            (strcmp(first->drain,second->drain)!=0)&&
            (strcmp(second->Type,first->Type)!=0)))
{
    found=TRUE;
}
else if(((strcmp(second->gate,first->gate))==0)&&
        (((((strcmp(second->source,first->drain))==0)&&
          (((strcmp(second->source,"Vdd")!=0)||
            (strcmp(second->source,"GND")!=0)))&&
            (((strcmp(second->drain,"Vdd")==0)&&
              (strcmp(first->source,"GND")==0))||
              (((strcmp(first->source,"Vdd")==0)&&
                (strcmp(second->drain,"GND")==0))))||
              (((strcmp(second->drain,first->source))==0)&&
                (((strcmp(second->drain,"Vdd")!=0)||
                  (strcmp(second->drain,"GND")!=0)))&&
                  (((strcmp(second->source,"Vdd")==0)&&
                    (strcmp(first->drain,"GND")==0))||
                    ((strcmp(first->drain,"Vdd")==0)&&
                     (strcmp(second->source,"GND")==0))))))&&
            (strcmp(second->Type,first->Type)!=0)))
{
    found=TRUE;
}
if(found==TRUE)
{
    /*****
    ** Increment header_invert->length which is the length for      **
    ** printing. Set done since an inverter has been found.          **
    *****/
    head_invert->length++;
    curr=head_invert->tail;
    done=TRUE;
    {
    /*****
    ** Create a node in the link list to hold the transistor        **

```



```

** information. **
*****/
    newinv = NewInvert();
/*****
** Is this the first entry on the list? If not, this is the **
** last entry. **
*****/
if (head_invert->head==NULL)
{
    head_invert->head=newinv;
    head_invert->tail=newinv;
}
else
{
    head_invert->tail=newinv;
    curr->next=newinv;
}
/*****
** Place the appropriate data into the inverter's fields. **
*****/
{
    newinv->gate=malloc(sizeof(char)*strlen(first->gate)+1);
    strcpy(newinv->gate,first->gate);
    if((strcmp(first->Type,"p")==0))
    {
        newinv->drain_ptype=malloc(sizeof(char)*strlen(first->drain)+1);
        strcpy(newinv->drain_ptype,first->drain);
        newinv->drain_ntype=malloc(sizeof(char)*strlen(second->drain)+1);
        strcpy(newinv->drain_ntype,second->drain);

        newinv->source_ptype=malloc(sizeof(char)*strlen(first->source)+1);
        strcpy(newinv->source_ptype,first->source);
        newinv->source_ntype=malloc(sizeof(char)*strlen(second->source)+1);
        strcpy(newinv->source_ntype,second->source);

        newinv->length_ptype=malloc(sizeof(char)*strlen(first->length)+1);
        strcpy(newinv->length_ptype,first->length);
        newinv->length_ntype=malloc(sizeof(char)*strlen(second->length)+1);
        strcpy(newinv->length_ntype,second->length);

        newinv->width_ptype=malloc(sizeof(char)*strlen(first->width)+1);
        strcpy(newinv->width_ptype,first->width);
        newinv->width_ntype=malloc(sizeof(char)*strlen(second->width)+1);
        strcpy(newinv->width_ntype,second->width);
    }
}

```

```

newinv->xloc_ptype=malloc(sizeof(char)*strlen(first->xloc)+1);
strcpy(newinv->xloc_ptype,first->xloc);
newinv->xloc_ntype=malloc(sizeof(char)*strlen(second->xloc)+1);
strcpy(newinv->xloc_ntype,second->xloc);

newinv->yloc_ptype=malloc(sizeof(char)*strlen(first->yloc)+1);
strcpy(newinv->yloc_ptype,first->yloc);
newinv->yloc_ntype=malloc(sizeof(char)*strlen(second->yloc)+1);
strcpy(newinv->yloc_ntype,second->yloc);
    }
    else
    {
newinv->drain_ntype=malloc(sizeof(char)*strlen(first->drain)+1);
strcpy(newinv->drain_ntype,first->drain);
newinv->drain_ptype=malloc(sizeof(char)*strlen(second->drain)+1);
strcpy(newinv->drain_ptype,second->drain);

newinv->source_ntype=malloc(sizeof(char)*strlen(first->source)+1);
strcpy(newinv->source_ntype,first->source);
newinv->source_ptype=malloc(sizeof(char)*strlen(second->source)+1);
strcpy(newinv->source_ptype,second->source);

newinv->length_ntype=malloc(sizeof(char)*strlen(first->length)+1);
strcpy(newinv->length_ntype,first->length);
newinv->length_ptype=malloc(sizeof(char)*strlen(second->length)+1);
strcpy(newinv->length_ptype,second->length);

newinv->width_ntype=malloc(sizeof(char)*strlen(first->width)+1);
strcpy(newinv->width_ntype,first->width);
newinv->width_ptype=malloc(sizeof(char)*strlen(second->width)+1);
strcpy(newinv->width_ptype,second->width);

newinv->xloc_ntype=malloc(sizeof(char)*strlen(first->xloc)+1);
strcpy(newinv->xloc_ntype,first->xloc);
newinv->xloc_ptype=malloc(sizeof(char)*strlen(second->xloc)+1);
strcpy(newinv->xloc_ptype,second->xloc);

newinv->yloc_ntype=malloc(sizeof(char)*strlen(first->yloc)+1);
strcpy(newinv->yloc_ntype,first->yloc);
newinv->yloc_ptype=malloc(sizeof(char)*strlen(second->yloc)+1);
strcpy(newinv->yloc_ptype,second->yloc);
    }
}

/*****
** Remove the transistors that now constitute an inverter.      **
** If the transistor lists' head or tail pointer are to be     **
*****/

```

```

** deleted change the head or tail. Delete the used transistors **
** and decrease the transistor lists' length. **
*****/
if((header_newp->length==1)&&(header_newn->length==1))
{
header_newn->length=0;
header_newp->length=0;
curr=head_invert->tail;
break;
}

    if(header_newn->tail==second)
    {
        header_newn->tail=prev2nd;
        prev2nd->next=NULL;
        complete=TRUE;
    }

    if(header_newn->head==second)
    {
        header_newn->head=second->next;
    }

    if((second!=prev2nd)&&(complete!=TRUE))
    {
        prev2nd->next=prev2nd->next->next;
        second=second->next;
    }

    else
    {
        prev2nd=prev2nd->next;
        second=second->next;
    }

    if(header_newp->tail==first)
    {
        stop=TRUE;
        header_newp->tail=prev1st;
        prev1st->next=NULL;
    }

    if(header_newp->head==first)
    {
        header_newp->head=first->next;
    }

    if((first!=prev1st)&&( stop!=TRUE))
    {
        prev1st->next=prev1st->next->next;
        first=first->next;
    }

```

```

        else
        {
previst=previst->next;
first=first->next;
}

        header_newn->length--;
        header_newp->length--;
        }
        }
        curr=head_invert->tail;
    }

/*****
** Is it a passgate?
*****/
else if((((strcmp(second->source,first->source))==0)&&
((strcmp(second->drain,first->drain))==0))||
(((strcmp(second->drain,first->source))==0)&&
((strcmp(second->source,first->drain))==0))&&
((strcmp(first->gate,"GND"))!=0)&&
((strcmp(second->gate,"GND"))!=0)&&
((strcmp(first->gate,"Vdd"))!=0)&&
((strcmp(second->gate,"Vdd"))!=0))
{
/*****
** Increment header_passgate->length which is the length for
** printing. Set done since an passgate has been found.
*****/
        head_passgate->length++;
        curr1=head_passgate->tail;
        done=TRUE;
        {
/*****
** Create a node in the link list to hold the transistor
** information.
*****/
        newpass = NewPass();
/*****
** Is this the first entry on the list? If not, this is the
** last entry.
*****/
if (head_passgate->head==NULL)
{
        head_passgate->head=newpass;
        head_passgate->tail=newpass;
}

```

```

else
{
    head_passgate->tail=newpass;
    curri->next=newpass;
}

/*****
** Place the appropriate data into the passgate's fields.      **
*****/

{
    newpass->drain=malloc(sizeof(char)*strlen(first->drain)+1);
    strcpy(newpass->drain,first->drain);
    if((strcmp(first->Type,"p")==0))
    {
        newpass->gate_ptype=malloc(sizeof(char)*strlen(first->gate)+1);
        strcpy(newpass->gate_ptype,first->gate);
        newpass->gate_ntype=malloc(sizeof(char)*strlen(second->gate)+1);
        strcpy(newpass->gate_ntype,second->gate);

        newpass->source_ptype=malloc(sizeof(char)*strlen(first->source)+1);
        strcpy(newpass->source_ptype,first->source);
        newpass->source_ntype=malloc(sizeof(char)*strlen(second->source)+1);
        strcpy(newpass->source_ntype,second->source);

        newpass->length_ptype=malloc(sizeof(char)*strlen(first->length)+1);
        strcpy(newpass->length_ptype,first->length);
        newpass->length_ntype=malloc(sizeof(char)*strlen(second->length)+1);
        strcpy(newpass->length_ntype,second->length);

        newpass->width_ptype=malloc(sizeof(char)*strlen(first->width)+1);
        strcpy(newpass->width_ptype,first->width);
        newpass->width_ntype=malloc(sizeof(char)*strlen(second->width)+1);
        strcpy(newpass->width_ntype,second->width);

        newpass->xloc_ptype=malloc(sizeof(char)*strlen(first->xloc)+1);
        strcpy(newpass->xloc_ptype,first->xloc);
        newpass->xloc_ntype=malloc(sizeof(char)*strlen(second->xloc)+1);
        strcpy(newpass->xloc_ntype,second->xloc);

        newpass->yloc_ptype=malloc(sizeof(char)*strlen(first->yloc)+1);
        strcpy(newpass->yloc_ptype,first->yloc);
        newpass->yloc_ntype=malloc(sizeof(char)*strlen(second->yloc)+1);
        strcpy(newpass->yloc_ntype,second->yloc);
    }
    else
    {

```

```

newpass->gate_ntype=malloc(sizeof(char)*strlen(first->gate)+1);
strcpy(newpass->gate_ntype,first->gate);
newpass->gate_ptype=malloc(sizeof(char)*strlen(second->gate)+1);
strcpy(newpass->gate_ptype,second->gate);

newpass->source_ntype=malloc(sizeof(char)*strlen(first->source)+1);
strcpy(newpass->source_ntype,first->source);
newpass->source_ptype=malloc(sizeof(char)*strlen(second->source)+1);
strcpy(newpass->source_ptype,second->source);

newpass->length_ntype=malloc(sizeof(char)*strlen(first->length)+1);
strcpy(newpass->length_ntype,first->length);
newpass->length_ptype=malloc(sizeof(char)*strlen(second->length)+1);
strcpy(newpass->length_ptype,second->length);

newpass->width_ntype=malloc(sizeof(char)*strlen(first->width)+1);
strcpy(newpass->width_ntype,first->width);
newpass->width_ptype=malloc(sizeof(char)*strlen(second->width)+1);
strcpy(newpass->width_ptype,second->width);

newpass->xloc_ntype=malloc(sizeof(char)*strlen(first->xloc)+1);
strcpy(newpass->xloc_ntype,first->xloc);
newpass->xloc_ptype=malloc(sizeof(char)*strlen(second->xloc)+1);
strcpy(newpass->xloc_ptype,second->xloc);

newpass->yloc_ntype=malloc(sizeof(char)*strlen(first->yloc)+1);
strcpy(newpass->yloc_ntype,first->yloc);
newpass->yloc_ptype=malloc(sizeof(char)*strlen(second->yloc)+1);
strcpy(newpass->yloc_ptype,second->yloc);
    }

/*****
** Remove the transistors that now constitute an passgate.      **
** If the transistor list's head or tail pointer are to be      **
** deleted change the head or tail. Delete the used transistors **
** and decrease the transistor lists' length.                    **
*****/
if((header_newp->length==1)&&(header_newn->length==1))
{
    header_newn->length=0;
    header_newp->length=0;
    curr1=head_passgate->tail;
    break;
}

    if(header_newn->tail==second)
{

```

```

    header_newn->tail=prev2nd;
    prev2nd->next=NULL;
    complete=TRUE;
}
    if(header_newn->head==second)
{
    header_newn->head=second->next;
}
    if((second!=prev2nd)&&(complete!=TRUE))
{
    prev2nd->next=prev2nd->next->next;
    second=second->next;
}
    else
{
    prev2nd=prev2nd->next;
    second=second->next;
}
    if(header_newp->tail==first)
{
    stop=TRUE;
    header_newp->tail=prev1st;
    prev1st->next=NULL;
}
    if(header_newp->head==first)
{
    header_newp->head=first->next;
}
    if((first!=prev1st)&&( stop!=TRUE))
{
    prev1st->next=prev1st->next->next;
    first=first->next;
}
    else
{
    prev1st=prev1st->next;
    first=first->next;
}
    header_newn->length--;
    header_newp->length--;
}
}
    curr1=head_passgate->tail;
}
/*****

```

```

** else go to the next transistor.                                     **
*****/
else
{
    prev2nd=second;
    second=second->next;
}
}

/*****
** go to the next transistor.                                         **
*****/
if((header_newp->length==0) || (header_newn->length==0))break;
{
    if(done!=TRUE)
    {
        prev1st=first;
        first=first->next;
    }
    second=header_newn->head;
    prev2nd=second;
}
}
}

```


E. LEVEL 1 RECOGNITION FUNCTION

```

/*****
** JOEL V. SWISHER          Thesis #2
** This program reads a.sim file as input. Then searches that
** file for possible level1 devices.
*****/

#include "rec.h"
level_one()
{
/*****
** The following pointers to trans are used to find level1
** devices.
*****/
trans *first, *prev1st, *second, *prev2nd;
/*****
** The following are used as flags when certain characteristics
** have been found for the transistors in level1 devices.
*****/
char c;
int nfound, pvdd;
/*****
** The following are counters.
*****/
int i;
first=header_newp->head;
prev1st=first;
nump=0;
numn=0;
numdevice=1;
numtrans=1;
ground=FALSE;
nfound=FALSE;
pvdd=FALSE;
while(first!=NULL)
{
/*****
** Is this the second time through on the same de
*****/
if(nfound==TRUE)
{
nfound=FALSE;
checkp(nump,numtrans-1);
}

```

```

/*****
** This is the first time through for this device.      **
** Find a transistor that is connected to Vdd.          **
*****/
else if(((strcmp(first->gate,"Vdd")==0)||
        ((strcmp(first->drain,"Vdd")==0)||
        ((strcmp(first->source,"Vdd")==0)))
{
    stacknum=1;
    Push(first);
    combinep(first,prev1st);
    pvdd=TRUE;
    nump=numtrans-1;
    nfound=FALSE;

/*****
** Find all of the n-type transistors that connect to the p-type**
** transistors already found.      **
*****/
for (i=1; i<=nump; i++)
{
    if(nfound==TRUE)break;
    second=header_newn->head;

/*****
** If there are no more n-type transistors then exit loop.      **
*****/
    if(second==NULL)break;
    prev2nd=second;

/*****
** Does the n-type transistor connect to the p-type transistor? **
*****/
    while((second!=NULL)&&(nfound==FALSE))
    {
        if(((strcmp(second->source,level1[numdevice][i]->source))==0)||
            ((strcmp(second->source,level1[numdevice][i]->drain))==0)||
            ((strcmp(second->drain,level1[numdevice][i]->source))==0)||
            ((strcmp(second->drain,level1[numdevice][i]->drain))==0))
        {
            stacknum=1;
            Push(second);
            combinen(second,prev2nd);
            nfound=TRUE;
        }
    }
else
{
    prev2nd=second;

```

```

second=second->next;
    }
}
}

/*****
** Has a p-type transistor which connects to Vdd been found?    **
** Yes, has a matching n-type been found? No, error in .sim      **
** file because a Vdd transistor must go to an n-type sometime. **
*****/
    if((pvdd==TRUE)&&(ground==FALSE))
    {
        tcount[numdevice]=numtrans-1;
        error2();
        printf("Do you want to quit and check your .sim file?(y or n)");
        c=getchar();
        while((c!='y')&&(c!='n'))
        {
            printf("(y or n)");
            c=getchar();
        }
        if(c=='y')
        {
            print_stats1();
            print_level1();
            exit(0);
        }
        else
        {
            ground=TRUE;
        }
    }

/*****
** Has a p-type transistor which connects to Vdd been found?    **
** No, continue the search through the linked list.             **
*****/
    else if(pvdd==FALSE)
    {
        prev1st=first;
        first=first->next;
    }

/*****
** A partial level1 device has been found. Repeat the loop and **
** see if there is another transistor in this device.          **
*****/

```

```

else if(nfound==TRUE)
{
first=header_newp->head;
prevlst=first;
}
/*****
** A level1 device has been found reset the search pointers and **
** see if there is another level1 device in the circuit.      **
*****/
    else
    {
tcount[numdevice]=numtrans-1;
numdevice++;
numtrans=1;
ground=FALSE;
nfound=FALSE;
pvdd=FALSE;
if(header_newp->length==0) break;
else
first=header_newp->head;
prevlst=first;
    }
}
/*****
** Determine whether above section was exited before numdevice **
** was incremented.
*****/
if(nfound==TRUE)
{
tcount[numdevice]=numtrans-1;
nfound=FALSE;
numdevice++;
}
/*****
** There are no transistors connected to Vdd left connect      **
** the remaining transistors in the file.                      **
*****/
{
/*****
** Combine the two transistor lists for easy recursive compares.**
*****/
header_new = create();
if(header_newp->head!=NULL)
{
header_new->head=header_newp->head;

```

```

header_newp->tail->next=header_newn->head;
}
else
{
header_new->head=header_newn->head;
}
header_new->tail=header_newn->tail;
first = header_new->head;
/*****
** There are no transistors connected to Vdd left connect      **
** the remaining transistors in the file.                      **
*****/
while(first!=NULL)
{
    stacknum=1;
    numtrans=1;
    previst=first;
    Push(first);
    combine(first,previst);
    first = header_new->head;
    tcount[numdevice]=numtrans-1;
    numdevice++;
}
}
/*****
** Prepare numdevice counter for printing.                    **
*****/
header_newp->length=0;
header_newn->length=0;
numdevice=numdevice-1;
}

/*****
** Function Push(T)                                           **
** This function places the transistor on the stack.         **
*****/

Push(T)
trans  *T;
{
    stack[stacknum] = T;
    if((strcmp(T->Type,"p"))==0)
    {
        pcount[numdevice]++;
    }
}

```

```

else
ncount[numdevice]++;
stacknum++;
}

/*****
** Function Pop()
** This function removes the transistor from the stack and
** places it into the level1 array.
*****/
Pop()
{
level1[numdevice][numtrans] = stack[--stacknum];
numtrans++;
}

/*****
** Function combine(start,prev1st)
** This function compares the transistor with all of the
** in the list to find a match. It calls itself recursively
** until all matches are found.
*****/

combine( start,prev1st)
trans *start, *prev1st;
{
trans *compare;
int set;
/*****
** Remove the transistor that now is part of a level 1 device.
** If the transistor lists' head or tail pointer are to be
** deleted change the head or tail. Delete the used transistor
** and decrease the transistor lists' length.
*****/
compare = start;
if(header_new->length == 1)
{
set = TRUE;
header_new->length = 0;
header_new->head = NULL;
start->next=NULL;
}
else if(start== header_new->head &&set!=TRUE)
{
header_new->head = start->next;

```

```

    header_new->length--;
}
else if(start== header_new->tail &&set!=TRUE)
{
    header_new->tail = previst;
    previst->next=NULL;
    header_new->length--;
}
else
{
    previst->next=previst->next->next;
    start->next=NULL;
    header_new->length--;
}
/*****
** Go to the head of the list to begin the comparisons.      **
*****/
if(header_new->head != NULL)
{
    start = header_new->head;
}
else start = NULL;
while(start != NULL)
{
/*****
** Are there any transistors that can be connected into a level1**
** device.                                                    **
*****/
    if(((strcmp(start->source,compare->source))==0)||
        ((strcmp(start->source,compare->drain))==0)||
        ((strcmp(start->drain,compare->source))==0)||
        ((strcmp(start->drain,compare->drain))==0))
        {
/*****
** A match was found, place the transistor on the stack and    **
** find any transistors which connect to it.                  **
*****/
            Push(start);
            combine(start,previst);
/*****
** The recursive call comes back to here. Reset the searching  **
** pointers to continue the search.                            **
*****/
            start=header_new->head;
        }
}

```

```

        else
        {
/*****
** No match was found increment the pointers.          **
*****/
        prev1st=start;
        start=start->next;
        }
    }
/*****
** No more transistors to compare this time. Place the      **
** into the level1 device array.                            **
*****/
Pop();
}

/*****
** Function combinep(start,prev1st)                        **
** This function compares the transistor with all of the      **
** in the list to find a match. It calls itself recursively  **
** until all matches are found.                             **
*****/

combinep( startp,prev1st)
trans *startp, *prev1st;
{
    trans *compare;
    int    set;
/*****
** Remove the transistor that now is part of a level 1 device. **
** If the transistor lists' head or tail pointer are to be    **
** deleted change the head or tail. Delete the used transistor **
** and decrease the transistor lists' length.                 **
*****/
    compare = startp;
    if(header_newp->length == 1)
    {
        set = TRUE;
        header_newp->length = 0;
        header_newp->head = NULL;
        startp->next=NULL;
    }
    else if(startp== header_newp->head && set!=TRUE)
    {
        header_newp->head = startp->next;
    }
}

```



```

        header_newp->length--;
    }
else if(startp== header_newp->tail &&set!=TRUE)
{
    header_newp->tail = prev1st;
    prev1st->next=NULL;
    header_newp->length--;
}
else
{
    prev1st->next=prev1st->next->next;
    startp->next=NULL;
    header_newp->length--;
}

/*****
** Go to the head of the list to begin the comparisons.      **
*****/
if(header_newp->head != NULL)
{
    startp = header_newp->head;
    prev1st=startp;
}
else startp = NULL;
while(startp != NULL)
{
/*****
** Are there any transistors that can be connected into a level1**
** device.                                                    **
*****/
    if((((strcmp(startp->source,compare->source))==0)&&
        ((strcmp(startp->source,"Vdd")!=0))||
        (((strcmp(startp->source,compare->drain))==0)&&
            ((strcmp(startp->source,"Vdd")!=0))||
            ((strcmp(startp->drain,compare->source))==0)&&
            ((strcmp(startp->drain,"Vdd")!=0))||
            (((strcmp(startp->drain,compare->drain))==0)&&
                ((strcmp(startp->drain,"Vdd")!=0))))
        {
/*****
** A match was found, place the transistor on the stack and    **
** find any transistors which connect to it.                  **
*****/
            Push(startp);
            combinep(startp,prev1st);
/*****

```

```

** The recursive call comes back to here. Reset the searching **
** pointers to continue the search. **
    compare=stack[stacknum];
    *****/
    startp=header_newp->head;
}
    else
    {
    /*****
    ** No match was found increment the pointers. **
    *****/
    previst=startp;
    startp=startp->next;
    }
}
/*****
** No more transistors to compare this time. Place the **
** into the level1 device array. **
*****/
Pop();
}

/*****
** Function combinen(startn,previst) **
** This function compares the transistor with all of the **
** in the list to find a match. It calls itself recursively **
** until all matches are found. **
*****/

combinen( startn,previst)
trans *startn, *previst;
{
    trans *compare;
    int set;
    compare = startn;
    /*****
    ** Does this n-type transistor connect to ground? **
    *****/
    if(((strcmp("GND",startn->source))==0)||
        ((strcmp("GND",startn->drain))==0))
        {
            ground=TRUE;
        }
    /*****
    ** Remove the transistor that now is part of a level 1 device. **

```

```

** If the transistor lists' head or tail pointer are to be      **
** deleted change the head or tail. Delete the used transistor  **
** and decrease the transistor lists' length.                    **
*****/
if(header_newn->length == 1)
{
    set = TRUE;
    header_newn->length = 0;
    header_newn->head = NULL;
    startn->next=NULL;
}
else if(startn== header_newn->head &&set!=TRUE)
{
    header_newn->head = startn->next;
    header_newn->length--;
}
else if(startn== header_newn->tail &&set!=TRUE)
{
    header_newn->tail = prev1st;
    prev1st->next=NULL;
    header_newn->length--;
}
else
{
    prev1st->next=prev1st->next->next;
    startn->next=NULL;
    header_newn->length--;
}
/*****
** Go to the head of the list to begin the comparisons.      **
*****/
if(header_newn->head != NULL)
{
    startn = header_newn->head;
    prev1st = startn;
}
else startn = NULL;
while(startn != NULL)
{
/*****
** Are there any transistors that can be connected into a level1**
** device.                                                         **
*****/
    if((((strcmp(startn->source,compare->source))==0)&&
        ((strcmp(startn->source,"GND"))!=0)))||

```

```

        (((strcmp(startn->source,compare->drain))==0)&&
        ((strcmp(startn->source,"GND"))!=0))||
        (((strcmp(startn->drain,compare->source))==0)&&
        ((strcmp(startn->drain,"GND"))!=0))||
        (((strcmp(startn->drain,compare->drain))==0)&&
        ((strcmp(startn->drain,"GND"))!=0)))
    {
/*****
** A match was found, place the transistor on the stack and      **
** find any transistors which connect to it.                      **
*****/
        Push(startn);
        combinen(startn,prev1st);
/*****
** The recursive call comes back to here. Reset the searching    **
** pointers to continue the search.                                **
*****/
        startn=header_newn->head;
    }
    else
    {
/*****
** No match was found increment the pointers.                      **
*****/
        prev1st=startn;
        startn=startn->next;
    }
}
/*****
** No more transistors to compare this time. Place the          **
** into the level1 device array.                                  **
*****/
Pop();
}

/*****
** Function checkp(lo,hi)                                          **
** This function compares the transistor with all of the         **
** in the list to find a match. It calls itself recursively      **
** until all matches are found.                                    **
*****/

checkp(lo,hi)
int lo,hi;
{

```

```

int pfound,nfound,i,j;
int hi2nd,lasthi;
trans *first,*prev1st,*second,*prev2nd;
nfound=FALSE;
first=header_newp->head;
prev1st=first;
second=header_newn->head;
prev2nd=second;
while(first!=NULL)
{
pfound=FALSE;
for(i=lo;i<=hi;i++)
{
if((((strcmp(first->source,level1[numdevice][i]->source))==0)&&
((strcmp(first->source,"Vdd"))!=0))||
(((strcmp(first->source,level1[numdevice][i]->drain))==0)&&
((strcmp(first->source,"Vdd"))!=0))||
(((strcmp(first->drain,level1[numdevice][i]->source))==0)&&
((strcmp(first->drain,"Vdd"))!=0))||
(((strcmp(first->drain,level1[numdevice][i]->drain))==0)&&
((strcmp(first->drain,"Vdd"))!=0)))
{
pfound=TRUE;
Push(first);
combinep(first,prev1st);
hi2nd=numtrans-1;
for(j=hi;j<=hi2nd;j++)
{
while(second!=NULL)
{
if((((strcmp(second->source,level1[numdevice][i]->source))==0)||
((strcmp(second->source,level1[numdevice][i]->drain))==0)||
((strcmp(second->drain,level1[numdevice][i]->source))==0)||
((strcmp(second->drain,level1[numdevice][i]->drain))==0))
{
stacknum=1;
Push(second);
combinen(second,prev2nd);
nfound=TRUE;
second=header_newn->head;
prev2nd=second;
lasthi=numtrans-1;
}
}
else
{

```

```

    prev2nd=second;
second=second->next;
    }
} /* end while second */
    } /* end for 2nd */
    } /* end if first */
if(pfound==TRUE)
{
    first=header_newp->head;
    prev1st=first;
    break;
}
} /* end for first */
if((pfound==FALSE)&&(first!=NULL))
{
    prev1st=first;
    first=first->next;
}
} /* end while first */
if(nfound==TRUE)
{
    checkp(hi2nd,lasthi);
}
}

```

F. LEVEL 2 RECOGNITION FUNCTION

```

/*****
** JOEL V. SWISHER      Thesis #2
** This program reads a.sim file as input. Then searches that
** file for possible level2 devices.
*****/

#include "rec.h"
level_two()
{
/*****
** The following pointers to inv and pass are used to find
** level2 devices.
*****/
    inv    *firstinv, *previnv;
    pass    *firstpass, *prevpass;
/*****
** The following are used as flags when certain characteristics
** have been found for the transistors in level2 devices.
*****/
    int out, found, outtest;
/*****
** The following are counters.
*****/
    int j, k, l, lz, first;
    vice=numdevice;
    nolvl=numdevice;
/*****
** Outtest checks to see if there is only one level1 device.
*****/
    outtest=numdevice+head_invert->length+head_passgate->length;
    lv2=1;
    first=FALSE;
    found=FALSE;
    out=FALSE;
/*****
** Fill the level2 device.
*****/
    while((nolvl> 0)|| (head_invert->length > 0)|| (head_passgate->length > 0))
    {
/*****
** Initialize array device counters.
*****/
        tcnt2[lv2]=0;

```

```

icnt2[lv2]=0;
pcnt2[lv2]=0;
/*****
** Initialize at least one level2 array.          **
*****/
if(nolv1>=1)
{
vice=nolv1;
first=TRUE;
fil_tran2();
}
else if(head_invert->length>=1)
{
first=TRUE;
firstinv = head_invert->head;
previnv = firstinv;
fil_inv2(previnv,firstinv);
}
else if(head_passgate->length>=1)
{
first=TRUE;
firstpass = head_passgate->head;
prevpass = firstpass;
fil_pass2(prevpass,firstpass);
}
else if( first==FALSE)
{
error_level2();
exit(0);
}
/*****
** Find any level1 devices that can be connected to this level2 **
** device. First compare level devices with level2 transistors. **
*****/
if((nolv1 > 0)&&(tcnt2[lv2]!=0))
{
for(j=1; j<=tcnt2[lv2]; j++)
{
vice=numdevice;
while(vice!=0)
{
for(k=1; k<=tcount[vice]; k++)
{
/*****
** Does the source match? It can not be Vdd or GND.          **
*****/

```



```

*****/
if((((strcmp(level2[lv2][j]->source,level1[vice][k]->source))==0)||
    ((strcmp(level2[lv2][j]->source,level1[vice][k]->drain))==0)||
    ((strcmp(level2[lv2][j]->source,level1[vice][k]->gate))==0))&&
    (((strcmp(level2[lv2][j]->source,"Vdd"))!=0)||
    ((strcmp(level2[lv2][j]->source,"GND"))!=0)))
{
    found=TRUE;
}
/*****/
** Does the drain match? It can not be Vdd or GND. **
*****/
else if((((strcmp(level2[lv2][j]->drain,level1[vice][k]->source))==0)||
    ((strcmp(level2[lv2][j]->drain,level1[vice][k]->gate))==0)||
    ((strcmp(level2[lv2][j]->drain,level1[vice][k]->drain))==0))&&
    (((strcmp(level2[lv2][j]->drain,"Vdd"))!=0)||
    ((strcmp(level2[lv2][j]->drain,"GND"))!=0)))
{
    found=TRUE;
}

/*****/
** Does the gate match? It can not be Vdd or GND. **
*****/
else if((((strcmp(level2[lv2][j]->gate,level1[vice][k]->source))==0)||
    ((strcmp(level2[lv2][j]->gate,level1[vice][k]->gate))==0)||
    ((strcmp(level2[lv2][j]->gate,level1[vice][k]->drain))==0))&&
    (((strcmp(level2[lv2][j]->gate,"Vdd"))!=0)||
    ((strcmp(level2[lv2][j]->gate,"GND"))!=0)))
{
    found=TRUE;
}

/*****/
** Does Level2[lv2][j] match? If so, has level1[vice][k] been **
** used yet? **
*****/
if((found==TRUE)&&((strcmp(level1[vice][k]->use_lv2,"F")==0))
{
    out=TRUE;
    fil_tran2();
    break; /* out of k for */
}
} /* close k for */

/*****/
** Reset found for next loop iteration. Decrement device num. **
*****/

```

```

found=FALSE;
    vice--;
    } /* close vice while */
    } /* close j for */
    } /* close if */
/*****
** Find any inverters that can be connected to this level2      **
** device. First compare inverters with level2 transistors.    **
*****/
    firstinv=head_invert->head;
    previnv=firstinv;
    if((head_invert->length>0)&&(tcnt2[lv2]!=0))
    {
while((firstinv!=NULL)&&(head_invert->length!=0))
{
    for(l=1; l<=tcnt2[lv2]; l++)
    {
/*****
** Does the source match? It can not be Vdd or GND.            **
*****/
        if((((strcmp(level2[lv2][l]->source,firstinv->source_ntype))==0)||
            ((strcmp(level2[lv2][l]->source,firstinv->drain_ntype))==0)||
            ((strcmp(level2[lv2][l]->source,firstinv->source_ptype))==0)||
            ((strcmp(level2[lv2][l]->source,firstinv->drain_ptype))==0)||
            ((strcmp(level2[lv2][l]->source,firstinv->gate))==0))&&
            (((strcmp(level2[lv2][l]->source,"Vdd"))!=0)||
             ((strcmp(level2[lv2][l]->source,"GND"))!=0))))
        {
            found=TRUE;
        }
/*****
** Does the drain match? It can not be Vdd or GND.            **
*****/
        else if((((strcmp(level2[lv2][l]->drain,firstinv->source_ntype))==0)||
            ((strcmp(level2[lv2][l]->drain,firstinv->source_ptype))==0)||
            ((strcmp(level2[lv2][l]->drain,firstinv->drain_ptype))==0)||
            ((strcmp(level2[lv2][l]->drain,firstinv->drain_ntype))==0)||
            ((strcmp(level2[lv2][l]->drain,firstinv->gate))==0))&&
            (((strcmp(level2[lv2][l]->drain,"Vdd"))!=0)||
             ((strcmp(level2[lv2][l]->drain,"GND"))!=0))))
        {
            found=TRUE;
        }
/*****
** Does the gate match? It can not be Vdd or GND.            **
*****/

```

```

*****/
else if((((strcmp(level2[lv2][1]->gate,firstinv->source_ntype))==0)||
        ((strcmp(level2[lv2][1]->gate,firstinv->source_ptype))==0)||
        ((strcmp(level2[lv2][1]->gate,firstinv->drain_ptype))==0)||
        ((strcmp(level2[lv2][1]->gate,firstinv->drain_ntype))==0)||
        ((strcmp(level2[lv2][1]->gate,firstinv->gate))==0))&&
        (((strcmp(level2[lv2][1]->gate,"Vdd"))!=0)||
         ((strcmp(level2[lv2][1]->gate,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Were there any matches? **
*****/
if(found==TRUE)
{
    out=TRUE;
    fil_inv2(previnv,firstinv);
    break;
}
} /* close for */
/*****
** Reset found for next loop iteration. Increment pointers. **
*****/
found=FALSE;
previnv=firstinv;
firstinv=firstinv->next;
} /* close while */
} /* close if */
/*****
** Find any passgates that can be connected to this level2 **
** device. First compare passgates with level2 transistors. **
*****/
    firstpass=head_passgate->head;
    prevpass=firstpass;
    if((head_passgate->length>0)&&(tcnt2[lv2]!=0))
    {
while((firstpass!=NULL)&&(head_passgate->length!=0))
{
    for(l=1; l<=tcnt2[lv2]; l++)
    {
/*****
** Does the source match? It can not be Vdd or GND. **
*****/
        if((((strcmp(level2[lv2][1]->source,firstpass->source_ntype))==0)||

```

```

        ((strcmp(level2[lv2][1]->source,firstpass->source_ptype))==0)||
        ((strcmp(level2[lv2][1]->source,firstpass->drain))==0)||
        ((strcmp(level2[lv2][1]->source,firstpass->gate_ntype))==0)||
        ((strcmp(level2[lv2][1]->source,firstpass->gate_ptype))==0))&&
        (((strcmp(level2[lv2][1]->source,"Vdd"))!=0)||
        ((strcmp(level2[lv2][1]->source,"GND"))!=0)))
    {
        found=TRUE;
    }
    /*****
    ** Does the drain match? It can not be Vdd or GND.
    *****/
    else if((((strcmp(level2[lv2][1]->drain,firstpass->source_ntype))==0)||
        ((strcmp(level2[lv2][1]->drain,firstpass->source_ptype))==0)||
        ((strcmp(level2[lv2][1]->drain,firstpass->drain))==0)||
        ((strcmp(level2[lv2][1]->drain,firstpass->gate_ntype))==0)||
        ((strcmp(level2[lv2][1]->drain,firstpass->gate_ptype))==0))&&
        (((strcmp(level2[lv2][1]->drain,"Vdd"))!=0)||
        ((strcmp(level2[lv2][1]->drain,"GND"))!=0)))
    {
        found=TRUE;
    }
    /*****
    ** Does the gate match? It can not be Vdd or GND.
    *****/
    else if((((strcmp(level2[lv2][1]->gate,firstpass->source_ntype))==0)||
        ((strcmp(level2[lv2][1]->gate,firstpass->source_ptype))==0)||
        ((strcmp(level2[lv2][1]->gate,firstpass->drain))==0)||
        ((strcmp(level2[lv2][1]->gate,firstpass->gate_ntype))==0)||
        ((strcmp(level2[lv2][1]->gate,firstpass->gate_ptype))==0))&&
        (((strcmp(level2[lv2][1]->gate,"Vdd"))!=0)||
        ((strcmp(level2[lv2][1]->gate,"GND"))!=0)))
    {
        found=TRUE;
    }
    /*****
    ** Were there any matches?
    *****/
    if(found==TRUE)
    {
        out=TRUE;
        fil_pass2(prevpass,firstpass);
        break;
    }
    } /* close for */

```

```

/*****
** Reset found for next loop iteration. Increment pointers.      **
*****/
found=FALSE;
prevpass=firstpass;
firstpass=firstpass->next;
    } /* close while */
} /* close if */
/*****
** Find any level1 devices that can be connected to this level2 **
** device. First compare level1 devices with level2 inverters. **
*****/
    if((nolvl1 > 0)&&(icnt2[lv2]!=0))
    {
for(j=1; j<=icnt2[lv2]; j++)
    {
        vice=numdevice;
        while(vice!=0)
        {
            for(k=1; k<=tcount[vice]; k++)
        {
/*****
** Does the ntype source match? It can not be Vdd or GND.      **
*****/
            if((((strcmp(ilevel2[lv2][j]->source_ntype,level1[vice][k]->source))==0)||
((strcmp(ilevel2[lv2][j]->source_ntype,level1[vice][k]->drain))==0)||
((strcmp(ilevel2[lv2][j]->source_ntype,level1[vice][k]->gate))==0))&&
(((strcmp(ilevel2[lv2][j]->source_ntype,"Vdd"))!=0)||
((strcmp(ilevel2[lv2][j]->source_ntype,"GND"))!=0)))
        {
            found=TRUE;
        }
/*****
** Does the ptype source match? It can not be Vdd or GND.      **
*****/
            else if((((strcmp(ilevel2[lv2][j]->source_ptype,level1[vice][k]->source))==0)||
((strcmp(ilevel2[lv2][j]->source_ptype,level1[vice][k]->drain))==0)||
((strcmp(ilevel2[lv2][j]->source_ptype,level1[vice][k]->gate))==0))&&
(((strcmp(ilevel2[lv2][j]->source_ptype,"Vdd"))!=0)||
((strcmp(ilevel2[lv2][j]->source_ptype,"GND"))!=0)))
        {
            found=TRUE;
        }
/*****
** Does the ntype drain match? It can not be Vdd or GND.      **

```

```

*****/
    else if((((strcmp(ilevel2[lv2][j]->drain_ntype,level1[vice][k]->source))==0)||
    ((strcmp(ilevel2[lv2][j]->drain_ntype,level1[vice][k]->drain))==0)||
    ((strcmp(ilevel2[lv2][j]->drain_ntype,level1[vice][k]->gate))==0))&&
    (((strcmp(ilevel2[lv2][j]->drain_ntype,"Vdd"))!=0)||
    ((strcmp(ilevel2[lv2][j]->drain_ntype,"GND"))!=0)))
    {
        found=TRUE;
    }
/*****/
** Does the ptype drain match? It can not be Vdd or GND.      **
*****/
    else if((((strcmp(ilevel2[lv2][j]->drain_ptype,level1[vice][k]->source))==0)||
    ((strcmp(ilevel2[lv2][j]->drain_ptype,level1[vice][k]->drain))==0)||
    ((strcmp(ilevel2[lv2][j]->drain_ptype,level1[vice][k]->gate))==0))&&
    (((strcmp(ilevel2[lv2][j]->drain_ptype,"Vdd"))!=0)||
    ((strcmp(ilevel2[lv2][j]->drain_ptype,"GND"))!=0)))
    {
        found=TRUE;
    }
/*****/
** Does the gate match? It can not be Vdd or GND.      **
*****/
    else if((((strcmp(ilevel2[lv2][j]->gate,level1[vice][k]->source))==0)||
    ((strcmp(ilevel2[lv2][j]->gate,level1[vice][k]->drain))==0)||
    ((strcmp(ilevel2[lv2][j]->gate,level1[vice][k]->gate))==0))&&
    (((strcmp(ilevel2[lv2][j]->gate,"Vdd"))!=0)||
    ((strcmp(ilevel2[lv2][j]->gate,"GND"))!=0)))
    {
        found=TRUE;
    }
/*****/
** Does Level2[lv2][j] match? If so, has level1[vice][k] been **
** used yet?                                                    **
*****/
    if((found==TRUE)&&((strcmp(level1[vice][k]->use_lv2,"F")==0))
    {
        out=TRUE;
        fil_tran2();
        break;
    }
    } /* close k for */
/***** *****/
** Reset found for next loop iteration. Decrement device num.  **
*****/

```

```

found=FALSE;
    vice--;
    } /* close vice while */
    } /* close j for */
    } /* close if */
/*****
** Find any inverters that can be connected to this level2      **
** device. Compare inverters with level2 inverters.             **
*****/
if((icnt2[lv2]!=0)&&(head_invert->length!=0))
{
    firstinv=head_invert->head;
    previnv=firstinv;
    if(head_invert->length>0)
    {
while((firstinv!=NULL)&&(head_invert->length!=0))
{
    for(lz=1; lz<=icnt2[lv2]; lz++)
    {
/*****
** Does the ntype source match? It can not be Vdd or GND.      **
*****/
        if((((strcmp(ilevel2[lv2][lz]->source_ntype,firstinv->source_ntype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ntype,firstinv->drain_ntype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ntype,firstinv->source_ptype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ntype,firstinv->drain_ptype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ntype,firstinv->gate))==0))&&
            (((strcmp(ilevel2[lv2][lz]->source_ntype,"Vdd"))!=0)||
             ((strcmp(ilevel2[lv2][lz]->source_ntype,"GND"))!=0)))
        {
            found=TRUE;
        }
/*****
** Does the ptype source match? It can not be Vdd or GND.      **
*****/
        else if((((strcmp(ilevel2[lv2][lz]->source_ptype,firstinv->source_ntype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ptype,firstinv->drain_ntype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ptype,firstinv->source_ptype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ptype,firstinv->drain_ptype))==0)||
            ((strcmp(ilevel2[lv2][lz]->source_ptype,firstinv->gate))==0))&&
            (((strcmp(ilevel2[lv2][lz]->source_ptype,"Vdd"))!=0)||
             ((strcmp(ilevel2[lv2][lz]->source_ptype,"GND"))!=0)))
        {
            found=TRUE;
        }
    }
}

```

```

/*****
** Does the ptype drain match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(ilevel2[lv2][lz]->drain_ptype,firstinv->source_ntype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ptype,firstinv->source_ptype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ptype,firstinv->drain_ptype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ptype,firstinv->drain_ntype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ptype,firstinv->gate))==0))&&
(((strcmp(ilevel2[lv2][lz]->drain_ptype,"Vdd"))!=0)||
((strcmp(ilevel2[lv2][lz]->drain_ptype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the ntype drain match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(ilevel2[lv2][lz]->drain_ntype,firstinv->source_ntype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ntype,firstinv->source_ptype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ntype,firstinv->drain_ptype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ntype,firstinv->drain_ntype))==0)||
((strcmp(ilevel2[lv2][lz]->drain_ntype,firstinv->gate))==0))&&
(((strcmp(ilevel2[lv2][lz]->drain_ntype,"Vdd"))!=0)||
((strcmp(ilevel2[lv2][lz]->drain_ntype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the gate match? It can not be Vdd or GND.             **
*****/
else if((((strcmp(ilevel2[lv2][lz]->gate,firstinv->source_ntype))==0)||
((strcmp(ilevel2[lv2][lz]->gate,firstinv->source_ptype))==0)||
((strcmp(ilevel2[lv2][lz]->gate,firstinv->drain_ptype))==0)||
((strcmp(ilevel2[lv2][lz]->gate,firstinv->drain_ntype))==0)||
((strcmp(ilevel2[lv2][lz]->gate,firstinv->gate))==0))&&
(((strcmp(ilevel2[lv2][lz]->gate,"Vdd"))!=0)||
((strcmp(ilevel2[lv2][lz]->gate,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Were there any matches?                                     **
*****/
if(found==TRUE)
{
    out=TRUE;
}

```



```

fil_inv2(previnv,firstinv);
break;
    }
} /* close for */
/*****
** Reset found for next loop iteration. Increment pointers.      **
*****/
    found=FALSE;
previnv=firstinv;
firstinv=firstinv->next;
    } /* close while */
} /* close if */
} /* close if */
/*****
** Find any passgates that can be connected to this level2      **
** device. Compare passgates with level2 inverters.             **
*****/
if((icnt2[lv2]!=0)&&(head_passgate->length>0))
{
    firstpass=head_passgate->head;
    prevpass=firstpass;
    if(head_passgate->length!=0)
    {
while((firstpass!=NULL)&&(head_passgate->length!=0))
{
    for(l=1; l<=icnt2[lv2]; l++)
    {
/*****
** Does the ntype source match? It can not be Vdd or GND.      **
*****/
        if((((strcmp(ilevel2[lv2][l]->source_ntype,firstpass->source_ntype))==0)||
            ((strcmp(ilevel2[lv2][l]->source_ntype,firstpass->source_ptype))==0)||
            ((strcmp(ilevel2[lv2][l]->source_ntype,firstpass->gate_ptype))==0)||
            ((strcmp(ilevel2[lv2][l]->source_ntype,firstpass->gate_ntype))==0)||
            ((strcmp(ilevel2[lv2][l]->source_ntype,firstpass->drain))==0))&&
            (((strcmp(ilevel2[lv2][l]->source_ntype,"Vdd"))!=0)||
            ((strcmp(ilevel2[lv2][l]->source_ntype,"GND"))!=0))))
        {
            found=TRUE;
        }
    }
/*****
** Does the ntype drain match? It can not be Vdd or GND.      **
*****/
    else if((((strcmp(ilevel2[lv2][l]->drain_ntype,firstpass->source_ntype))==0)||
            ((strcmp(ilevel2[lv2][l]->drain_ntype,firstpass->source_ptype))==0)||

```

```

                ((strcmp(ilevel2[lv2][1]->drain_ntype,firstpass->gate_ptype))==0)||
                ((strcmp(ilevel2[lv2][1]->drain_ntype,firstpass->gate_ntype))==0)||
                ((strcmp(ilevel2[lv2][1]->drain_ntype,firstpass->drain))==0))&&
                (((strcmp(ilevel2[lv2][1]->drain_ntype,"Vdd"))!=0)||
                ((strcmp(ilevel2[lv2][1]->drain_ntype,"GND"))!=0)))
        {
                found=TRUE;
        }
}
/*****
** Does the ptype source match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(ilevel2[lv2][1]->source_ptype,firstpass->source_ntype))==0)||
        ((strcmp(ilevel2[lv2][1]->source_ptype,firstpass->source_ptype))==0)||
        ((strcmp(ilevel2[lv2][1]->source_ptype,firstpass->gate_ptype))==0)||
        ((strcmp(ilevel2[lv2][1]->source_ptype,firstpass->gate_ntype))==0)||
        ((strcmp(ilevel2[lv2][1]->source_ptype,firstpass->drain))==0))&&
        (((strcmp(ilevel2[lv2][1]->source_ptype,"Vdd"))!=0)||
        ((strcmp(ilevel2[lv2][1]->source_ptype,"GND"))!=0)))
        {
                found=TRUE;
        }
}
/*****
** Does the ptype drain match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(ilevel2[lv2][1]->drain_ptype,firstpass->source_ntype))==0)||
        ((strcmp(ilevel2[lv2][1]->drain_ptype,firstpass->source_ptype))==0)||
        ((strcmp(ilevel2[lv2][1]->drain_ptype,firstpass->gate_ptype))==0)||
        ((strcmp(ilevel2[lv2][1]->drain_ptype,firstpass->gate_ntype))==0)||
        ((strcmp(ilevel2[lv2][1]->drain_ptype,firstpass->drain))==0))&&
        (((strcmp(ilevel2[lv2][1]->drain_ptype,"Vdd"))!=0)||
        ((strcmp(ilevel2[lv2][1]->drain_ptype,"GND"))!=0)))
        {
                found=TRUE;
        }
}
/*****
** Does the gate match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(ilevel2[lv2][1]->gate,firstpass->source_ntype))==0)||
        ((strcmp(ilevel2[lv2][1]->gate,firstpass->source_ptype))==0)||
        ((strcmp(ilevel2[lv2][1]->gate,firstpass->gate_ptype))==0)||
        ((strcmp(ilevel2[lv2][1]->gate,firstpass->gate_ntype))==0)||
        ((strcmp(ilevel2[lv2][1]->gate,firstpass->drain))==0))&&
        (((strcmp(ilevel2[lv2][1]->gate,"Vdd"))!=0)||
        ((strcmp(ilevel2[lv2][1]->gate,"GND"))!=0)))
        {

```

```

    found=TRUE;
}
/*****
** Were there any matches? **
*****/
if(found==TRUE)
{
    out=TRUE;
    fil_pass2(prevpass,firstpass);
    break;
}
} /* close for */
/*****
** Reset found for next loop iteration. Increment pointers. **
*****/
    found=FALSE;
prevpass=firstpass;
firstpass=firstpass->next;
} /* close while */
} /* close if */
} /* close if */

/*****
** Find any level1 devices that can be connected to this level2 **
** device. First compare level1 devices with level2 passgates. **
*****/
    if((nolvl > 0)&&(pcnt2[lv2]!=0))
    {
        for(j=1; j<=pcnt2[lv2]; j++)
        {
            vice=numdevice;
            while(vice!=0)
            {
                for(k=1; k<=tcount[vice]; k++)
                {
/*****
** Does the ntype source match? It can not be Vdd or GND. **
*****/
                    if((((strcmp(plevel2[lv2][j]->source_ntype,level1[vice][k]->source))==0)||
                        ((strcmp(plevel2[lv2][j]->source_ntype,level1[vice][k]->drain))==0)||
                        ((strcmp(plevel2[lv2][j]->source_ntype,level1[vice][k]->gate))==0))&&
                        (((strcmp(plevel2[lv2][j]->source_ntype,"Vdd")!=0)||
                          ((strcmp(plevel2[lv2][j]->source_ntype,"GND")!=0))))
                    {
                        found=TRUE;
                    }
                }
            }
        }
    }
}

```

```

/*****
** Does the ptype source match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][j]->source_ptype,level1[vice][k]->source))==0)||
        ((strcmp(plevel2[lv2][j]->source_ptype,level1[vice][k]->drain))==0)||
        ((strcmp(plevel2[lv2][j]->source_ptype,level1[vice][k]->gate))==0))&&
        (((strcmp(plevel2[lv2][j]->source_ptype,"Vdd")!=0)||
        ((strcmp(plevel2[lv2][j]->source_ptype,"GND")!=0))))
{
    found=TRUE;
}
/*****
** Does the ptype gate match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][j]->gate_ptype,level1[vice][k]->source))==0)||
        ((strcmp(plevel2[lv2][j]->gate_ptype,level1[vice][k]->drain))==0)||
        ((strcmp(plevel2[lv2][j]->gate_ptype,level1[vice][k]->gate))==0))&&
        (((strcmp(plevel2[lv2][j]->gate_ptype,"Vdd")!=0)||
        ((strcmp(plevel2[lv2][j]->gate_ptype,"GND")!=0))))
{
    found=TRUE;
}
/*****
** Does the ntype gate match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][j]->gate_ntype,level1[vice][k]->source))==0)||
        ((strcmp(plevel2[lv2][j]->gate_ntype,level1[vice][k]->drain))==0)||
        ((strcmp(plevel2[lv2][j]->gate_ntype,level1[vice][k]->gate))==0))&&
        (((strcmp(plevel2[lv2][j]->gate_ntype,"Vdd")!=0)||
        ((strcmp(plevel2[lv2][j]->gate_ntype,"GND")!=0))))
{
    found=TRUE;
}
/*****
** Does the drain match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][j]->drain,level1[vice][k]->source))==0)||
        ((strcmp(plevel2[lv2][j]->drain,level1[vice][k]->drain))==0)||
        ((strcmp(plevel2[lv2][j]->drain,level1[vice][k]->gate))==0))&&
        (((strcmp(plevel2[lv2][j]->drain,"Vdd")!=0)||
        ((strcmp(plevel2[lv2][j]->drain,"GND")!=0))))
{
    found=TRUE;
}
/*****

```

```

** Does Level2[lv2][j] match? If so, has level1[vice][k] been    **
** used yet?                                                    **
*****/
    if((found==TRUE)&&((strcmp(level1[vice][k]->use_lv2,"F")==0))
        {
out=TRUE;
fil_tran2();
break;
    }
} /* close k for */
/*****
** Reset found for next loop iteration. Decrement device num.  **
*****/
    found=FALSE;
    vice--;
} /* close vice while */
    } /* close j for */
    } /* close if */
/*****
** Find any inverters that can be connected to this level2      **
** device. Compare inverters with level2 passgates.             **
*****/
if((pcnt2[lv2]!=0)&&(head_invert->length>0))
{
    firstinv=head_invert->head;
    previnv=firstinv;
    if(head_invert->length!=0)
    {
while((firstinv!=NULL)&&(head_invert->length!=0))
{
    for(lz=1; lz<=icnt2[lv2]; lz++)
    {
/*****
** Does the ntype source match? It can not be Vdd or GND.      **
*****/
        if((((strcmp(plevel2[lv2][lz]->source_ntype,firstinv->source_ntype))==0)||
            ((strcmp(plevel2[lv2][lz]->source_ntype,firstinv->drain_ntype))==0)||
            ((strcmp(plevel2[lv2][lz]->source_ntype,firstinv->source_ptype))==0)||
            ((strcmp(plevel2[lv2][lz]->source_ntype,firstinv->drain_ptype))==0)||
            ((strcmp(plevel2[lv2][lz]->source_ntype,firstinv->gate))==0))&&
            (((strcmp(plevel2[lv2][lz]->source_ntype,"Vdd"))!=0)||
            ((strcmp(plevel2[lv2][lz]->source_ntype,"GND"))!=0)))
        {
            found=TRUE;
        }
    }
}

```

```

/*****
** Does the ptype source match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][lz]->source_ptype,firstinv->source_ntype))==0)||
((strcmp(plevel2[lv2][lz]->source_ptype,firstinv->drain_ntype))==0)||
((strcmp(plevel2[lv2][lz]->source_ptype,firstinv->source_ptype))==0)||
((strcmp(plevel2[lv2][lz]->source_ptype,firstinv->drain_ptype))==0)||
((strcmp(plevel2[lv2][lz]->source_ptype,firstinv->gate))==0))&&
(((strcmp(plevel2[lv2][lz]->source_ptype,"Vdd"))!=0)||
((strcmp(plevel2[lv2][lz]->source_ptype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the ptype gate match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][lz]->gate_ptype,firstinv->source_ntype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ptype,firstinv->drain_ntype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ptype,firstinv->source_ptype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ptype,firstinv->drain_ptype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ptype,firstinv->gate))==0))&&
(((strcmp(plevel2[lv2][lz]->gate_ptype,"Vdd"))!=0)||
((strcmp(plevel2[lv2][lz]->gate_ptype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the ntype gate match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][lz]->gate_ntype,firstinv->source_ntype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ntype,firstinv->drain_ntype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ntype,firstinv->source_ptype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ntype,firstinv->drain_ptype))==0)||
((strcmp(plevel2[lv2][lz]->gate_ntype,firstinv->gate))==0))&&
(((strcmp(plevel2[lv2][lz]->gate_ntype,"Vdd"))!=0)||
((strcmp(plevel2[lv2][lz]->gate_ntype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the drain match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][lz]->drain,firstinv->source_ntype))==0)||
((strcmp(plevel2[lv2][lz]->drain,firstinv->source_ptype))==0)||
((strcmp(plevel2[lv2][lz]->drain,firstinv->drain_ptype))==0)||

```

```

        ((strcmp(plevel2[lv2][lz]->drain,firstinv->drain_ntype))==0)||
        ((strcmp(plevel2[lv2][lz]->drain,firstinv->gate))==0))&&
        (((strcmp(plevel2[lv2][lz]->drain,"Vdd")!=0)||
        ((strcmp(plevel2[lv2][lz]->drain,"GND")!=0)))
    {
        found=TRUE;
    }
    /*****
    ** Were there any matches?
    *****/
    if(found==TRUE)
    {
        out=TRUE;
        fil_inv2(previnv,firstinv);
        break;
    }
    } /* close for */
    /*****
    ** Reset found for next loop iteration. Increment pointers.
    *****/
    found=FALSE;
    previnv=firstinv;
    firstinv=firstinv->next;
    } /* close while */
    } /* close if */
    } /* close if */
    /*****
    ** Find any passgates that can be connected to this level2
    ** device. Compare passgates with level2 passgates.
    *****/
    if((pcnt2[lv2]!=0)&&(head_passgate->length>0))
    {
        firstpass=head_passgate->head;
        prevpass=firstpass;
        if(head_passgate->length!=0)
        {
            while((firstpass!=NULL)&&(head_passgate->length!=0))
            {
                for(l=1; l<=pcnt2[lv2]; l++)
                {
                    /*****
                    ** Does the ntype source match? It can not be Vdd or GND.
                    *****/
                    if((((strcmp(plevel2[lv2][l]->source_ntype,firstpass->source_ntype))==0)||
                    ((strcmp(plevel2[lv2][l]->source_ntype,firstpass->source_ptype))==0)||

```

```

((strcmp(plevel2[lv2][1]->source_ntype,firstpass->gate_ntype))==0)||
((strcmp(plevel2[lv2][1]->source_ntype,firstpass->gate_ptype))==0)||
    ((strcmp(plevel2[lv2][1]->source_ntype,firstpass->drain))==0))&&
    (((strcmp(plevel2[lv2][1]->source_ntype,"Vdd"))!=0)||
    ((strcmp(plevel2[lv2][1]->source_ntype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the ptype source match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][1]->source_ptype,firstpass->source_ntype))==0)||
    ((strcmp(plevel2[lv2][1]->source_ptype,firstpass->source_ptype))==0)||
    ((strcmp(plevel2[lv2][1]->source_ptype,firstpass->gate_ntype))==0)||
    ((strcmp(plevel2[lv2][1]->source_ptype,firstpass->gate_ptype))==0)||
    ((strcmp(plevel2[lv2][1]->source_ptype,firstpass->drain))==0))&&
    (((strcmp(plevel2[lv2][1]->source_ptype,"Vdd"))!=0)||
    ((strcmp(plevel2[lv2][1]->source_ptype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the ptype gate match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][1]->gate_ptype,firstpass->source_ntype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ptype,firstpass->source_ptype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ptype,firstpass->gate_ntype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ptype,firstpass->gate_ptype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ptype,firstpass->drain))==0))&&
    (((strcmp(plevel2[lv2][1]->gate_ptype,"Vdd"))!=0)||
    ((strcmp(plevel2[lv2][1]->gate_ptype,"GND"))!=0)))
{
    found=TRUE;
}
/*****
** Does the ntype gate match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][1]->gate_ntype,firstpass->source_ntype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ntype,firstpass->source_ptype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ntype,firstpass->gate_ntype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ntype,firstpass->gate_ptype))==0)||
    ((strcmp(plevel2[lv2][1]->gate_ntype,firstpass->drain))==0))&&
    (((strcmp(plevel2[lv2][1]->gate_ntype,"Vdd"))!=0)||
    ((strcmp(plevel2[lv2][1]->gate_ntype,"GND"))!=0)))
{

```



```

    found=TRUE;
}

/*****
** Does the drain match? It can not be Vdd or GND.      **
*****/
else if((((strcmp(plevel2[lv2][1]->drain,firstpass->source_ntype))==0)||
((strcmp(plevel2[lv2][1]->drain,firstpass->source_ptype))==0)||
((strcmp(plevel2[lv2][1]->drain,firstpass->gate_ntype))==0)||
((strcmp(plevel2[lv2][1]->drain,firstpass->gate_ptype))==0)||
((strcmp(plevel2[lv2][1]->drain,firstpass->drain))==0))&&
(((strcmp(plevel2[lv2][1]->drain,"Vdd"))!=0)||
((strcmp(plevel2[lv2][1]->drain,"GND"))!=0)))
{
    found=TRUE;
}

/*****
** Were there any matches?      **
*****/
if(found==TRUE)
{
    out=TRUE;
    fil_pass2(prevpass,firstpass);
    break;
}

} /* close for */

/*****
** Reset found for next loop iteration. Increment pointers.      **
*****/
    found=FALSE;
    prevpass=firstpass;
    firstpass=firstpass->next;
} /* close while */
} /* close if */
} /* close if */

/*****
** Has any matches been found(out==TRUE)? if yes, repeat      **
** procedure, else increment level2 device counter.      **
*****/
outtest=nolvi+head_invert->length+head_passgate->length;
    if(out==TRUE)
    {
        out=FALSE;
    }

    else if(outtest>=1)
    {

```

```

        lv2++;
    }
}

/*****
** End While exit
*****/

/*****
** Function fil_tran2()
** This function fills the transistor array for level2 elements.
*****/

fil_tran2()
{
    int i;
    for(i=1; i<= tcount[vice]; i++)
    {
        tcnt2[lv2]++;
        level2[lv2][tcnt2[lv2]]=level1[vice][i];
        level1[vice][i]->use_lv2="T";
    }
    nolv1--;
}

/*****
** Function fil_inv2()
** This function fills the inverter array for level2 elements.
*****/

fil_inv2(p,f)
inv *p,*f;
{
    int stop;
    stop=FALSE;
    icnt2[lv2]++;
    ilevel2[lv2][icnt2[lv2]] = f;
    /*****
    ** Remove the inverter that now is part of a level 2 device.
    ** If the inverter lists' head or tail pointer are to be
    ** deleted change the head or tail. Delete the used inverter
    ** and decrease the inverter lists' length.
    *****/
    if(head_invert->length==1)
    {

```

```

        head_invert->head=NULL;
        head_invert->length=0;
        stop=TRUE;
    }
    if((head_invert->tail==f)&&(stop!=TRUE))
    {
        stop=TRUE;
        head_invert->tail=p;
        p->next=NULL;
    }
    if(head_invert->head==f)
    {
        head_invert->head=f->next;
    }
    if((f!=p)&&( stop!=TRUE))
    {
        p->next=p->next->next;
        f=f->next;
    }
    else if(stop!=TRUE)
    {
        p=p->next;
        f=f->next;
    }
    head_invert->length--;
}

/*****
** Function fil_pass2()
** This function fills the passgate array for level2 elements.
** *****/

fil_pass2(p,f)
pass *p,*f;
{
    int stop;
    stop=FALSE;
    pcnt2[lv2]++;
    plevel2[lv2][pcnt2[lv2]] = f;
/*****
** Remove the passgate that now is part of a level 2 device.
** If the passgate lists' head or tail pointer are to be
** deleted change the head or tail. Delete the used passgate
** and decrease the passgate lists' length.
** *****/

```

```

if(head_passgate->length==1)
{
    head_passgate->head=NULL;
    head_passgate->length=0;
    stop=TRUE;
}
if((head_passgate->tail==f)&&(stop!=TRUE))
{
    stop=TRUE;
    head_passgate->tail=p;
    p->next=NULL;
}
if(head_passgate->head==f)
{
    head_passgate->head=f->next;
}
if((f!=p)&&( stop!=TRUE))
{
    p->next=p->next->next;
    f=f->next;
}
else if(stop!=TRUE)
{
    p=p->next;
    f=f->next;
}
head_passgate->length--;
}

```

G. PRINT FUNCTIONS

```

/*****
** Function print_ptrans()
** To print the list of ptransistors.
*****/
#include "rec.h"

print_ptrans()
{
    int i;
    trans *node;
    node = header_newp->head;
    if(header_newp->length!=0)
    {
        for(i=1; i<= header_newp->length; i++)
        {
            fprintf(fo,"P-type Gate %d is:   %s \n",
                i, (node->gate));
            fprintf(fo,"Source %d is:   %s \n", i, (node->source));
            fprintf(fo,"Drain %d is:   %s \n", i, (node->drain));
            node = (node->next);
        }
    }
}

/*****
** Function print_ntrans()
** To print the list of ntransistors.
*****/

print_ntrans()
{
    int i;
    trans *node;
    node = header_newn->head;
    if(header_newn->length!=0)
    {
        for(i=1; i<= header_newn->length; i++)
        {
            fprintf(fo,"N-type Gate %d is:   %s \n",
                i, (node->gate));
            fprintf(fo,"Source %d is:   %s \n", i, (node->source));
            fprintf(fo,"Drain %d is:   %s \n", i, (node->drain));
            node = (node->next);
        }
    }
}

```

```

}
}
}

/*****
** Function print_invert()                               **
** To print the list of invertors.                         **
** #include "rec.h"                                         **
*****/

print_invert()
{
    int i;
    inv *node_inv;
    node_inv = head_invert->head;
    for(i=1; i<= head_invert->length; i++)
    {
        fprintf(fo,"The inverter gate %d is:   %s \n",
                i, (node_inv->gate));
        fprintf(fo,"The inverter p-type drain %d is:   %s \n",
                i, (node_inv->drain_ptype));
        fprintf(fo,"The inverter n-type drain %d is:   %s \n",
                i, (node_inv->drain_ntype));
        fprintf(fo,"The inverter p-type source %d is:   %s \n",
                i, (node_inv->source_ptype));
        fprintf(fo,"The inverter n-type source %d is:   %s \n",
                i, (node_inv->source_ntype));
        node_inv = (node_inv->next);
    }
}

/*****
** Function print_pass()                                   **
** To print the list of passgate.                         **
** #include "rec.h"                                         **
*****/

print_pass()
{
    int i;
    pass *node_pass;
    node_pass = head_passgate->head;
    for(i=1; i<= head_passgate->length; i++)
    {
        fprintf(fo,"The passgate drain %d is:   %s \n",

```

```

        i, (node_pass->drain));
fprintf(fo,"The passgate p-type source %d is:  %s \n",
        i, (node_pass->source_ptype));
fprintf(fo,"The passgate n-type source %d is:  %s \n",
        i, (node_pass->source_ntype));
fprintf(fo,"The passgate p-type gate %d is:   %s \n",
        i, (node_pass->gate_ptype));
fprintf(fo,"The passgate n-type gate %d is:   %s \n",
        i, (node_pass->gate_ntype));
node_pass = (node_pass->next);
}
}

/*****
** Function print_stats1()
** To print the list of the initial statistics.
*****/

print_stats1()
{
fprintf(fo,"\n\nThe Scale in centrimicrons is %s ",scale);
fprintf(fo,"and %s is the technology used.\n\n",technology);
fprintf(fo,"This circuit has a total of %d transistor%c.\n\n",
total_transistors,(total_transistors==1) ? ' ' : 's');
fprintf(fo,"This circuit has %d inverter%c.\n",
total_invert,(total_invert==1) ? ' ' : 's');
fprintf(fo,"This circuit has %d passgate%c.\n",
total_passgates,(total_passgates==1) ? ' ' : 's');
fprintf(fo,"This circuit has %d other level1 device%c.\n\n",
numdevice,(numdevice==1) ? ' ' : 's');
print_invert();
print_pass();
}

/*****
** Function print_level1()
** To print the list of level1 devices.
*****/

print_level1()
{
    int i,j;
    trans *node;
fprintf(fo,"\n\n*****Other Level1 Devices*****\n");
    for(i=1; i<= numdevice; i++)

```

```

    {
        fprintf(fo, "\nThere are %d transistors in this level1 device.\n", tcount[i]);
        for(j=1; j<= tcount[i]; j++)
        {
            node = level1[i] [j];
            fprintf(fo, "%s-type Gate %d is:   %s \n", node->Type,
j, (node->gate));
            fprintf(fo, "Source %d is:   %s \n", j, (node->source));
            fprintf(fo, "Drain %d is:   %s \n", j, (node->drain));
        }
    }
}

```

```

/*****
** Function print_stats2()
** To print the list of final statisitics.
**
*****/

```

```

print_stats2()
{
    int i;
    fprintf(fo, "\n*****Level2 Devices*****\n");
    fprintf(fo, "This circuit has %d level2 device%c.\n",
lv2, (lv2==1) ? ' ' : 's');
    for(i=1; i<= lv2; i++)
    {
        fprintf(fo, "\n Level2 device number %d has: \n", i);
        fprintf(fo, " %d transistor%c, ", tcnt2[i], (tcnt2[i]==1) ? ' ' : 's');
        fprintf(fo, " %d inverter%c, ", icnt2[i], (icnt2[i]==1) ? ' ' : 's');
        fprintf(fo, " and %d passgate%c. \n", pcnt2[i], (pcnt2[i]==1) ? ' ' : 's');
    }
}

```


H. FUNCTION TO READ THE INPUT FILE

```

/*****
** JOEL V. SWISHER          Filler.c                      **
** This program fills the buffer with the next data from the **
** input file.                      **
*****/
#include "rec.h"
filler()
{
    int    complete, done;
    register int  c;
    /* (storage class) c is an integer stored in a register */
    strcpy(buffer,""); /* string copy initializes buffer */
    len = 0;
    done=FALSE;
    while(!done)
    {
        c= getc(fp); /* c equals next character in argv[1] */
        if(newline(c)) Check=TRUE;
        if (c==EOF)
        {
            printf("end of the file encountered\n");
            fprintf (fo,"end of file encountered\n");
            complete=TRUE;
            print_ptrans();
            print_ntrans();
            print_invert();
            print_pass();
            break; /* exits for loop */
        }
        if ( (' '< c && c<0175))
        { /* Is c a writeable char? */
            buffer[len++]=c; /* Yes, fill buffer. */
        }
        if (IsWhite(c) && (buffer[0] != ' '))
        /* If c is a blank, tab, or newline and buffer is not empty */
        {
            buffer[len] = '\0'; /* insert end of string marker */
            done=TRUE;
        }
        /* get the next word (continue the FOR stmt) */
    }
    if(complete==TRUE) exit(1);
}

```

I. FUNCTIONS TO CREATE DATA STRUCTURES

```

/*****
** Function create()
** Creates the head and tail pointer node called temp.
**
*****/
#include "rec.h"

head_type *create()
{
head_type *temp;
temp = MALLOC(head_type);
temp->length = 0;
temp->head = temp->tail = NULL;
return(temp);
}

/*****
** Function createinv()
** Creates the head and tail pointer node for the invertor
** called tmp.
**
*****/
#include "rec.h"

head_inv *createinv()
{
head_inv *tmp;
tmp = MALLOC(head_inv);
tmp->length = 0;
tmp->head = tmp->tail = NULL;
return(tmp);
}

/*****
** Function createpass()
** Creates the head and tail pointer node for the passgate
** called tp.
**
*****/
#include "rec.h"

head_pass *createpass()
{
head_pass *tp;
tp = MALLOC(head_pass);
tp->length = 0;
tp->head = tp->tail = NULL;
return(tp);
}

```

```

}
/*****
** Function NewNode()                               **
** Prepare a new trans node for the new device      **
*****/
#include "rec.h"

trans *NewNode()
{
    trans *newnode;
    if(!(newnode = MALLOC(trans)))
    {
        printf("out of the storage \n");
        exit(1);
    }
    newnode->next=NULL;
    newnode->Type=NULL;
    newnode->gate=NULL;
    newnode->source=NULL;
    newnode->drain=NULL;
    newnode->length=NULL;
    newnode->width=NULL;
    newnode->xloc=NULL;
    newnode->yloc=NULL;
    newnode->use_lv2="F";
    return(newnode);
}
/*****
** Function NewInvert()                             **
** Prepare a new inverter node for the new inverter. **
*****/
#include "rec.h"

inv *NewInvert()
{
    inv *newinvert;
    if(!(newinvert = MALLOC(inv)))
    {
        printf("out of the storage \n");
        exit(1);
    }
    newinvert->gate=NULL;
    newinvert->source_ptype=NULL;
    newinvert->source_ntype=NULL;
    newinvert->drain_ptype=NULL;

```

```

newinvert->drain_ntype=NULL;
newinvert->length_ptype=NULL;
newinvert->length_ntype=NULL;
newinvert->width_ptype=NULL;
newinvert->width_ntype=NULL;
newinvert->xloc_ptype=NULL;
newinvert->xloc_ntype=NULL;
newinvert->yloc_ptype=NULL;
newinvert->yloc_ntype=NULL;
newinvert->next=NULL;
return(newinvert);
}

/*****
** Function NewPass()
** Prepare a new passgate node for the new passgate.
**
#include "rec.h"
*****/

pass *NewPass()
{
pass *newpass;
if(!(newpass = MALLOC(pass)))
{
printf("out of the storage \n");
exit(1);
}
newpass->gate_ptype=NULL;
newpass->gate_ntype=NULL;
newpass->source_ptype=NULL;
newpass->source_ntype=NULL;
newpass->drain=NULL;
newpass->length_ptype=NULL;
newpass->length_ntype=NULL;
newpass->width_ptype=NULL;
newpass->width_ntype=NULL;
newpass->xloc_ptype=NULL;
newpass->xloc_ntype=NULL;
newpass->yloc_ptype=NULL;
newpass->yloc_ntype=NULL;
newpass->next=NULL;
return(newpass);
}

```

J. ADDITIONAL FUNCTIONS

```

/*****
** JOEL V. SWISHER      Error.c      **
** This program provides the error message for the program.      **
*****/

#include "rec.h"
error()
{
printf("Improper simulation file. This program will work when used\n");
printf("inconjunction with a file created from the \"ext2sim\" command\n");
printf("which is part of the University of California Berkley (UCB)\n");
printf("tools package.\n");
}

/*****
** JOEL V. SWISHER      error2()      **
** This function provides the error message for the level1      **
** procedure.          **
*****/

error2()
{
printf("Improper simulation file. This program will work when used\n");
printf("inconjunction with a file created from the \"ext2sim\" command\n");
printf("which is part of the University of California Berkley (UCB)\n");
printf("tools package.\n\n");
printf("There is a problem in the file in that a transistor connected\n");
printf("to Vdd is never connected to GRN. Please verify your circuit.\n");
fprintf(fo,"Improper simulation file. This program will work when used\n");
fprintf(fo,"inconjunction with a file created from the \"ext2sim\" command\n");
fprintf(fo,"which is part of the University of California Berkley (UCB)\n");
fprintf(fo,"tools package.\n\n");
fprintf(fo,"There is a problem in the file in that a transistor connected\n");
fprintf(fo,"to Vdd is never connected to GRN. Please verify your circuit.\n");
}

error_level2()
{
printf("There are no level1 devices. This is not possible if there \n");
printf("were any transistors in the .sim file.\n");
}

```

REFERENCES

1. Bose, A.K., "Progress in VLSI Design Automation," *Tenth European Solid-State Circuits Conference*, September 1984, pp. 103-104.
2. Jacobs, H., "Verification of a Second Generation 32-bit Microprocessor," *Computer*, April 1986, pp. 64-70.
3. Vladimirescu, A., Kaihe Zhang, A.R. Newton, D.O. Pederson, and H. Sangiovanni-Vincentelli, *SPICE User's Guide*, University of California Berkeley, February 1987.
4. Corbin, L.V., "Custom VLSI Electrical Rule Checking in an Intelligent Terminal," *Proceedings of the 18th ACM/IEEE Design Automation Conference*, June 1981, pp. 696-701.
5. Ablasser, I., and U. Jager, "Circuit Recognition and Verification Based on Layout Information," *Proceedings of the 18th ACM/IEEE Design Automation Conference*, June 1981, pp. 684-689.
6. Ng, Pauline, Wolfram Glauert, and Robert Kirk, "A Timing Verification System Based on Extracted MOS/VLSI Circuit Parameters," *Proceedings of the 18th ACM/IEEE Design Automation Conference*, June 1981, pp. 288-292.
7. Hitchcock, Robert B. Sr., "Timing Verification and the Timing Analysis Program," *Proceedings of the 19th ACM/IEEE Design Automation Conference*, June 1982, pp. 594-604.
8. Ousterhout, John K., "A Switch-Level Timing Verifier for Digital MOS VLSI," *IEEE Transactions on Computer-Aided Design*, July 1985, pp. 336-349.
9. Hamachi, Gordon T., and John K. Ousterhout, "A Switchbox Router with Obstacle Avoidance," *Proceedings of the 21st ACM/IEEE Design Automation Conference*, June 1984, pp. 173-179.
10. Ousterhout, John K., Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor, "Magic: A VLSI Layout System," *Proceedings of the 21st ACM/IEEE Design Automation Conference*, June 1984, pp. 152-165.
11. Scott, Walter S., and John K. Ousterhout, "Plowing: Interactive Stretching and Compaction in Magic," *Proceedings of the 21st ACM/IEEE Design Automation Conference*, June 1984, pp. 166-172.
12. Scott, Walter S., and John K. Ousterhout, "Magic's Circuit Extractor," *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, June 1985, pp. 286-292.

13. University of California Berkeley Report No UCB/CSD 86/272, *1986 VLSI Tools: Still More Works by the Original Artists; "Berkeley CAD Tools User's Manual"* by Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, December 1986.
14. Carleton, David A., *A Pad Frame Generator with Automatic Routing for VLSI Chip Assembly*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

INITIAL DISTRIBUTION LIST

| | No. of Copies |
|--|---------------|
| 1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145 | 2 |
| 2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002 | 2 |
| 3. Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000 | 1 |
| 4. Superintendent, Naval Postgraduate School Attn: Professor Chyan Yang, Code 62Ya Naval Postgraduate School Monterey, California 93943-5000 | 6 |
| 5. Superintendent, Naval Postgraduate School Attn: Professor Jon T. Butler, Code 62Bu Naval Postgraduate School Monterey, California 93943-5000 | 2 |
| 6. Commander USAISEC Attn: ASB-SEP-C Building 31043 Fort Huachuca, Arizona 85613-5300 | 1 |
| 7. Naval Research Laboratory Code 9110-52, LT Brian Kosinski 4555 Overlook Avenue Southwest Washington, D.C. 20375 | 1 |

8. CPT Joel V. Swisher
HHC, USAISEC
Fort Huachuca, Arizona 85613

1